

# 现代前端 技术解析

张成文 编著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

## 内 容 简 介

这是一本以现代前端技术思想与理论为主要内容的书。前端技术发展迅速，涉及的技术点很多，我们往往需要阅读很多书籍才能理解前端技术的知识体系。这本书在前端知识体系上做了很好的总结和梳理，涵盖了现代前端技术绝大部分的知识内容，起到一个启蒙作用，能帮助读者快速把握前端技术的整个脉络，培养更完善的体系化思维，掌握更多灵活的前端代码架构方法，使读者获得成为高级前端工程师或架构师所必须具备的思维和能力。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有，侵权必究。

## 图书在版编目（CIP）数据

现代前端技术解析 / 张成文编著. —北京：电子工业出版社，2017.4  
ISBN 978-7-121-31033-1

I. ①现… II. ①张… III. ①网页制作工具—研究 IV. ①TP393.092.2

中国版本图书馆 CIP 数据核字(2017)第 043435 号

责任编辑：徐津平

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：21.25 字数：445 千字

版 次：2017 年 4 月第 1 版

印 次：2017 年 4 月第 1 次印刷

印 数：3000 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式：010-51260888-819，[faq@phei.com.cn](mailto:faq@phei.com.cn)。

# 前言

欢迎阅读本书，本书是一本以现代前端技术思想与理论为主要内容的书。前端技术发展迅速，涉及的技术点很多，我们往往需要阅读很多本书籍才能理解前端技术的知识体系。这本书在前端知识体系上为大家做了很好的总结和梳理，涵盖了现代前端技术绝大部分的知识内容，包括前端技术基础、开发调试技术、前端相关协议、三层结构演进与实践、响应式网站、页面交互框架、大型项目实践经验、跨栈开发实践等，这些都能使大家获得成为高级前端工程师或架构师所必须具备的思维和能力。

## 目标读者

本书主要面向各类前端工程师。

初中级前端工程师们可以通过本书快速地领略到前端领域的深度和广度，把握整个前端技术领域的发展方向和所涵盖的绝大多数技术要点，为未来深入学习前端知识提供指导和方向。当然，笔者还是建议你在有一定前端技术基础的前提下再来阅读本书，因为这并不是一本入门的工具类书籍，对基础知识讲解较少。如果你已经是高级前端工程师，也希望你能通过本书了解到自己在前端知识体系中没有掌握的内容。笔者欢迎你对本书的内容提出建议，帮助我指出书中的不足，补充书中没有涉及的内容。

## 写作目的

本书的写作目的是希望通过笔者对现代前端知识体系的剖析来帮助那些需要快速成长和提升的前端读者们，为他们提供一个系统的知识体系和明确的学习方向，快速了解现代前端所涉及的主要知识点，把握前端知识体系结构的整个脉络。前端入门很简单，但是优秀的前端工程师却很少，如果你希望变得更优秀，那你一定会需要这本书。如果你觉得自己已经很优秀，也

可以阅读本书进一步梳理知识体系，帮助自己查缺补漏。

不希望你购买本书后因为工作忙而翻几页便将它一直放在某个角落，所以，本书会用尽量少的篇幅和浅显的语句让大家掌握最核心的技术思想，少数偏理论的细节不会展开太多，只将其中最关键的部分讲清楚。如果大家希望有一本更全面、更清晰的书籍，笔者后期也会在本书的基础上深入展开。

总之，本书的写作目的是，希望区别于基础入门的工具书而从实质上帮助所有的前端读者们快速理解前端技术知识体系，培养自身更完善的体系化思维，掌握更多灵活的前端代码架构技术。最后，衷心希望这本书能真正帮助需要知识引导的同学，起到一个学习启蒙的作用，也希望大家多多支持这本书。

## 写作背景

随着互联网的持续发展和网络信息的多样化，我们对网络信息的获取量越来越大，获取方式越来越多，获取媒介种类也越来越复杂。就目前而言，我们生活的方方面面都与互联网息息相关，通信、娱乐、购物、企业服务等均已成为每天生活中不可或缺的部分。就获取信息的媒介来看，终端设备（主要包括个人电脑、智能手机、平板电脑等）是我们从互联网上获取信息的最主要媒介。

与此同时，互联网信息展现的内容和形式越来越偏向于终端设备屏幕，而且基于终端设备的交互越来越多，越来越复杂。目前在终端设备屏幕上，获取互联网信息的最主要途径仍然是通过 Web 浏览器（或内嵌浏览器 WebView，下文中统称为 Web 浏览器），网络信息在 Web 浏览器上是以 Web 应用的形式展现的。随着信息量的增大和信息多样化增加，Web 浏览器应用也越来越复杂，规模越来越庞大，原有的由后台服务器直接产生网页数据的技术已经不能满足需要。

在我国，随着 2000 年左右第一批互联网企业的崛起，中国互联网累积输出的网络信息量越来越大，用户数越来越多，信息业务也越来越多样化。国内许多第一批从事网站开发的技术人员渐渐意识到 Web 浏览器端业务信息处理逻辑的复杂，因此他们不能把主要精力只集中在服务器端的数据收集和处理工作上。任何一项复杂工作简单化的最好方式必然是分工，在这种情况下，第一批半职业的前端工程师出现了，具体时间可以大致认为是 Web 2.0 时代开始的时候，也就是说，国内的前端工程师可以认为是在以用户原创产生内容（User Generated Content, UGC）为主的互联网应用网站产生时开始出现的。不仅如此，在 UGC 应用大行其道的时代，各类电子商务网站的蓬勃发展又大大增加了互联网企业对前端工程师的需求。

随着互联网的普及，社交和电商的蓝海中涌现出了无数的互联网企业，这类企业中的大部分目前仍然是以利用 Web 浏览器与用户进行信息交互为主要业务，用来完成用户的信息消费。自 2008 年开始，移动互联网的另一波浪潮让互联网行业的发展如日中天，整个国内外互联网行业一片欣欣向荣。与此同时，为数不多的前端工程师和呈指数增长的前端工程师需求量形成了巨大的产业人力资源矛盾。不少互联网公司因为业务发展需要，鼓励大量服务端工程师向前端工程师转型来填补前端人力的空缺。直至今今天，前端工程师的数量仍然远远不能满足企业的发展需要，不过与 UGC 时代相比，前端工程师的数量有了一定的提升。与此同时，互联网应用场景的复杂化也提高了企业对前端工程师基本能力的要求，一部分初级前端工程师仍然不能胜任企业的工作，而优秀的前端工程师一将难求。

面对这种形势，笔者觉得，目前缺乏优秀前端工程师的主要因素有以下两点。

### 📌 前端专业教育引导资源的稀缺

可以认为，前端工程师培养起来难度大，和前端教育学习资源的缺乏有一定的关系。虽然有专门的前端方向培训机构，但是这种模式下的人才培养专业度和产出完全不如人意。笔者的感觉是，一般毕业就被选拔进入大型互联网公司且技术能力相对较强的前端工程师都是通过平时自学进入这一领域的。笔者很明白前端学习的困难，所以如果有充足的前端教育资源来引导，结果就有可能不一样。

### 📌 前端领域的技术革新速度快，对前端工程师的要求越来越高

真正了解前端技术的工程师都会感觉前端技术发展变化远快于其他端。浏览器特性、编程语言标准、前端框架、前端工具、多终端浏览器等都在快速地换代更新。作为一名前端工程师，不仅要掌握现有的技术来实现业务需求，解决业务问题，还要不断快速学习新的技术知识，为新技术时代的到来做准备。对于后接触的人来说，需要了解掌握的东西会越来越多。

所以希望本书所讲解的前端体系化内容能够降低前端从业人员的学习成本，帮助读者快速从宏观上把握前端知识体系结构的整个脉络。

## 主要内容

本书一共分 7 章，每一章的主要内容如下所述。

- 第 1 章，主要介绍现代 Web 应用的发展概况和相关的技术知识，同时也深入总结目前主要浏览器的基础原理知识与常用的前端开发调试技术。

- 第2章，主要讲解了前端技术的相关协议，包括 HTTP、HTTP2、HTTPS、Web 实时协议、前端安全机制、RESTful 规范和 Hybrid 混合应用交互协议等。
- 第3章，以前端的三层结构发展演进和实践技术为主，讲解 HTML5、ECMAScript 5+、CSS3 的发展历程和特性，同时介绍现代前端开发中的编译技术和响应式站点的设计思路。
- 第4章，主要讲解前端交互框架的演进和各类前端框架的实现原理，包括直接型 DOM 交互框架、MVC、MVP、MVVM、Virtual DOM、MNV\*等框架的设计思路。
- 第5章，以专题的方式讲解现代前端大型项目的实践思路，主要包括开发规范、组件化规范、构建原理、用户数据分析、前端优化手段、搜索引擎优化基础等内容。
- 第6章，围绕跨栈主题介绍前端技术栈在后端和移动端 Hybrid 上的开发实践，例如，后端直出同构原理、Hybrid 离线包与增量机制实现等关键架构的设计思路。
- 第7章，就前端的未来趋势进行分析，简单介绍物联网 Web 化、Web VR、网站人工智能等未来的前端技术趋势，总结成为一名优秀前端工程师的要素。

相信读者们看到这里就会兴奋起来，因为本书涵盖了现代前端中大部分需要掌握的技术实践理念，这也是编写本书的初衷。相比于深入介绍具体某一个前端框架的书籍，这本书能带给读者体系化思维和技术理念上的提升，因为前端学习不是学会某个框架就可以了。

## 写作声明

关于本书的内容，在此声明如下。

1. 书中所涉及的内容均为基于笔者理解之上的原著输出，部分内容如有雷同，属于共性知识。
2. 书中提出了一些自定义概念，可能之前没有被提出，请读者认真理解。
3. 书中涉及技术原理类的内容较多，并不是只介绍某个特定框架的工具类书籍，建议读者在有一定前端技术基础的前提下阅读，本书的编写宗旨是提高前端工程师的体系思维和设计能力，而不是帮大家快速入门。
4. 本书使用的全部样例代码均基于 ECMAScript 6+环境，需要较高版本的浏览器或 Node

服务器的运行支持。

5. 本书涉及的知识点量大，涵盖范围广，对相关基础知识细节的展开不会过于详细，但对于每种技术的实现都有较深入的原理性分析，希望读者认真阅读领会。

## 致谢

感谢为了这本书得以完成而做出贡献的小伙伴们。首先感谢电子工业出版社策划编辑陈晓猛的辛苦跟进和鼎力协助，保证了书籍能够按期出版；其次感谢腾讯 IMWEB 团队的培养，没有团队给予的锻炼机会，我根本不可能完成这本书；最后感谢腾讯 IMWEB 团队的成员们帮我提供素材和校对稿件；另外要特别感谢来自腾讯的 kenkozheng、samarali、helondeng 和来自携程的 sheiladai 等同事给予本书的写作建议与技术审校。由衷感谢大家。

张成文

---

## 读者服务

轻松注册成为博文视点社区用户（[www.broadview.com.cn](http://www.broadview.com.cn)），您即可享受以下服务：

- **提交勘误** 您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流** 在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31033>

二维码：



# 专家推荐

近几年前端技术发展迅猛，各大公司对前端优秀人才的需求急剧增加。本书从一名一线专业前端从业者的角度，面面俱到地为大家剖析了当前 Web 前端所需要具备的各种现代技术。无论是从网络、浏览器方面，还是从工程化、团队协作的角度都给出了非常好的呈现，非常值得大家阅读。

郭学亨(Henry)，腾讯前端 IMWeb 团队负责人

近几年，有关前端的书籍很少能全面而且深入地介绍前端技术思想与理论相关内容，大部分都是独立拆分介绍前端单点领域的技术栈。这本书以现代前端技术思想与理论为主，详细而且深入，但又通俗地向读者阐述了现代前端技术栈，无论对初学者还是中级学者都是值得一读的好书。读者可以通过本书快速领略到前端领域的深度和广度，把握整个前端技术领域所涵盖的绝大多数知识技术要点和发展方向，为未来深入学习前端知识提供指导和方向。

大漠，W3cplus.com 站长

现如今，前端已经不再是一种“新兴职业”，对技术系统且全面的追求愈显重要，但繁杂的技术体系及各种旁支经常让初学者无所适从。本书能从全局和主流的视野介绍前端职业工程师几乎涉猎的所有知识，并将前端工作中涉及的解决方案分门别类，抽象成易于理解的思路。相信对前端感兴趣的读者一定能够借助这本难得的好书触类旁通，一帆风顺地推开通往前端界的神秘大门，快速地成为一名优秀的前端工程师。

许诺 (Darksnow)，阿里巴巴前端无线开发专家



本书从一名前端工程师的角度，梳理了现代前端技术所涉及的基础知识体系和原理性技术解析，包括开发方式的变更、前端框架的演进、前端跨栈技术以及未来的 VR 等，契合当前流行的“大前端”概念，非常适合读者扩宽个人知识面。另外，作者本人在前端方面有很深的造诣，对目前的一些前端问题有深入的研究和个人独特的见解。相信读者朋友们一定能从本书中收获颇多。

邓海龙(Helon)，腾讯前端 IMWeb 团队成员

前端技术日新月异且涉及的知识面较广泛，对于初学者来说又不知从何学起、所学的东西是否已经过时。对于中级学者，可能又存在对某些知识的了解不够全面和深刻的问题。本书由前端技术的发展历程向整体架构体系逐层展开，基本涵盖了现阶段的前端技能树，为前端学习者指明了方向。同时，本书注重从原理的层面进行知识点的解析，万变不离其宗，各种框架或技术之间的很多思想是相通的，把握住这一点，将对读者后续的学习更有帮助。

李双双(Lissa)，腾讯前端工程师



# 目录

第 1 章 Web 前端技术基础 .....	1
1.1 现代 Web 前端技术发展概述 .....	1
1.1.1 现代 Web 前端技术应用 .....	1
1.1.2 现代 Web 前端技术概述 .....	4
1.1.3 Web 前端技术发展 .....	6
1.2 浏览器应用基础 .....	10
1.2.1 浏览器组成结构 .....	10
1.2.2 浏览器渲染引擎简介 .....	12
1.2.3 浏览器数据持久化存储技术 .....	20
1.3 前端高效开发技术 .....	34
1.3.1 前端高效开发工具 .....	34
1.3.2 前端高效调试工具 .....	36
1.4 本章小结 .....	42
第 2 章 前端与协议 .....	43
2.1 HTTP 协议简介 .....	43
2.1.1 HTTP 协议概述 .....	43
2.1.2 HTTP 1.1 .....	45
2.1.3 HTTP 2 .....	51
2.2 web 安全机制 .....	53
2.2.1 基础安全知识 .....	53
2.2.2 请求劫持与 HTTPS .....	57
2.2.3 HTTPS 协议通信过程 .....	59

2.2.4	HTTPS 协议解析 .....	61
2.2.5	浏览器 Web 安全控制 .....	63
2.3	前端实时协议.....	64
2.3.1	WebSocket 通信机制.....	65
2.3.2	Poll 和 Long-poll .....	66
2.3.3	前端 DDP 协议.....	70
2.4	RESTful 数据协议规范 .....	71
2.5	与 Native 交互协议.....	73
2.5.1	Hybrid App 应用概述.....	74
2.5.2	Web 到 Native 协议调用 .....	74
2.5.3	Native 到 Web 协议调用 .....	77
2.5.4	JSBridge 设计规范 .....	78
2.6	本章小结.....	81
第 3 章	前端三层结构与应用 .....	82
3.1	HTML 结构层基础 .....	83
3.1.1	必须要知道的 DOCTYPE .....	83
3.1.2	Web 语义化标签 .....	84
3.1.3	HTML 糟糕的部分 .....	87
3.1.4	AMP HTML.....	90
3.2	前端结构层演进.....	94
3.2.1	XML 与 HTML 简述 .....	94
3.2.2	HTML5 标准 .....	95
3.2.3	HTML Web Component.....	96
3.3	浏览器脚本演进历史.....	102
3.3.1	CoffeeScript 时代 .....	103
3.3.2	ECMAScript 标准概述.....	105
3.3.3	TypeScript 概况 .....	105
3.3.4	JavaScript 衍生脚本 .....	106
3.4	JavaScript 标准实践.....	107
3.4.1	ECMAScript 5.....	107
3.4.2	ECMAScript 6.....	113

---

3.4.3	ECMAScript 7+ .....	128
3.4.4	TypeScript .....	130
3.5	前端表现层基础.....	131
3.5.1	CSS 发展概述.....	131
3.5.2	CSS 选择器与属性.....	132
3.5.3	简单的应用举例.....	133
3.6	前端界面技术.....	135
3.6.1	CSS 样式统一化.....	136
3.6.2	CSS 预处理.....	138
3.6.3	表现层动画实现.....	141
3.6.4	CSS4 与展望.....	149
3.7	响应式网站开发技术.....	149
3.7.1	响应式页面实现概述.....	149
3.7.2	结构层响应式.....	152
3.7.3	表现层响应式.....	160
3.7.4	行为层响应式.....	166
3.8	本章小结.....	167
<b>第 4 章</b>	<b>现代前端交互框架 .....</b>	<b>168</b>
4.1	直接 DOM 操作时代 .....	168
4.2	MV*交互模式 .....	176
4.2.1	前端 MVC 模式.....	176
4.2.2	前端 MVP 模式 .....	180
4.2.3	前端 MVVM 模式.....	181
4.2.4	数据变更检测示例 .....	185
4.3	Virtual DOM 交互模式 .....	193
4.3.1	Virtual DOM 设计理念 .....	193
4.3.2	Virtual DOM 的核心实现 .....	196
4.4	前端 MNV*时代 .....	200
4.4.1	MNV*模式简介 .....	201
4.4.2	MNV*模式实现原理 .....	201
4.5	本章小结.....	203

第 5 章 前端项目与技术实践 .....	204
5.1 前端开发规范.....	204
5.1.1 前端通用规范.....	205
5.1.2 前端 HTML 规范 .....	208
5.1.3 前端 CSS 规范.....	212
5.1.4 ECMAScript 5 常用规范.....	218
5.1.5 ECMAScript 6+ 参考规范 .....	222
5.1.6 前端防御性编程规范.....	227
5.2 前端组件规范.....	229
5.2.1 UI 组件规范.....	230
5.2.2 模块化规范.....	233
5.2.3 项目组件化设计规范.....	237
5.3 自动化构建.....	242
5.3.1 自动化构建的目的 .....	243
5.3.2 自动化构建原理.....	243
5.3.3 构建工具设计的问题.....	246
5.4 前端性能优化.....	248
5.4.1 前端性能测试.....	248
5.4.2 桌面浏览器前端优化策略.....	253
5.4.3 移动端浏览器前端优化策略.....	258
5.5 前端用户数据分析.....	266
5.5.1 用户访问统计.....	266
5.5.2 用户行为分析.....	267
5.5.3 前端日志上报.....	270
5.5.4 前端性能分析上报.....	275
5.6 前端搜索引擎优化基础.....	275
5.6.1 title、keywords、description 的优化 .....	275
5.6.2 语义化标签的优化.....	277
5.6.3 URL 规范化.....	278
5.6.4 robots.....	279
5.6.5 sitemap .....	279

5.7	前端协作.....	280
5.7.1	沟通能力和沟通技巧.....	280
5.7.2	与产品经理的“对抗”.....	281
5.7.3	与后台工程师的合作.....	281
5.7.4	与运维工程师的“周旋”.....	282
5.7.5	对前端团队的支持.....	282
5.8	本章小结.....	283
第 6 章	前端跨栈技术.....	284
6.1	JavaScript 跨后端实现技术.....	284
6.1.1	Node 后端开发基础概述.....	285
6.1.2	早期 MEAN 简介.....	288
6.1.3	Node 后端数据渲染.....	289
6.1.4	前后端同构概述.....	290
6.1.5	前后端同构实现原理.....	291
6.2	跨终端设计与实现.....	297
6.2.1	Hybrid 技术趋势.....	297
6.2.2	Hybrid 实现方式.....	299
6.2.3	基于 localStorage 的资源离线和更新技术.....	301
6.2.4	基于 Native 与 Web 的资源离线和更新技术.....	308
6.2.5	资源覆盖率统计.....	310
6.2.6	仍需要注意的问题.....	311
6.3	本章小结.....	312
第 7 章	未来前端时代.....	313
7.1	未来前端趋势.....	314
7.1.1	新标准的进化与稳定.....	314
7.1.2	应用开发技术趋于稳定并将等待下一次革新.....	314
7.1.3	持续不断的技术工具探索.....	315
7.1.4	浏览器平台新特性的应用.....	315
7.1.5	更优化的前端技术开发生态.....	315
7.1.6	前端新领域的出现.....	316

7.2 做一名优秀的前端工程师 .....	318
7.2.1 学会高效沟通.....	318
7.2.2 使用高效的开发工具.....	319
7.2.3 处理问题方法论.....	319
7.2.4 学会前端项目开发流程设计.....	320
7.2.5 持续的知识 and 经验积累管理.....	321
7.2.6 切忌过分追求技术.....	321
7.2.7 必要的产品设计思维.....	322
7.3 本章小结.....	323



# 第 1 章

## Web 前端技术基础

前端技术自诞生以来，就一直保持着较快的发展速度。与此同时，前端技术的快速发展对前端工程师的要求也越来越高。到目前为止，除了浏览器的应用复杂度在提升，前端的开发模式也在不断演进。前端开发模式先后经历了静态黄页时期、服务器组装动态网页数据时期、后端为主的 MVC（Model-View-Controller，一种数据模型、视图、逻辑分离的开发模式）模式时期、前后端分离方案开发时期、纯前端 MV\*（Model-View-\*, 数据模型、视图、控制方式分离的开发设计方式，这里的控制方式有多种实现方法，所以一般用\*代替）为主与中间层直出的时期，最后进入到前端 Virtual DOM（虚拟 DOM，用来描述页面 DOM 树节点之间关系的一种特殊 JavaScript 对象）、MNV\*（Model-NativeView-\*, 数据模型、原生视图、控制方式分离的开发设计模式，这里的控制方式也可以有多种实现方法，所以用\*代替）模式以及前后端同构的开发时代。可能你对这些内容还不熟悉，不过不用担心，在后面的章节中将会详细为大家介绍。

工欲善其事，必先利其器。为了更好地学习和了解后面的知识，我们有必要先来了解现代 Web 前端技术的发展概况和 Web 浏览器的基础开发技术。在这一章中，我们就先来了解一下 Web 应用技术基础及其发展概况。

### 1.1 现代Web前端技术发展概述

#### 1.1.1 现代Web前端技术应用

互联网信息呈现的方式越来越偏向于终端设备屏幕，而且终端设备屏幕上的交互也越来越多，越来越复杂。如图 1-1 所示，如今我们获取网络信息的设备种类越来越多，除了借助传统的个人计算机来访问网络，还可以通过智能手机、平板电脑或穿戴设备等来获取网络信息。



图 1-1 现代网络信息主要访问媒介

而在众多不同终端设备屏幕上，获取互联网信息的最主要途径仍然是 Web 浏览器（内嵌浏览器 WebView，后面统称为 Web 浏览器），网络信息内容在 Web 浏览器上最终是以 Web 应用的形式展现的。

如今，互联网 Web 浏览器上的应用已经变得十分庞杂，接下来我们将通过一个具体的例子来了解现代 Web 浏览器应用的具体发展是怎么样的。

图 1-2 是目前一个非常典型的浏览器端 Web 应用——腾讯课堂的首页部分用户界面。注意，这只是一部分，整个页面的内容很长很复杂，可操作的部分也很多，我们只截取了头部导航的一部分内容进行分析。从用户界面上来看，这部分其实已经包含了很多的数据内容和用户交互，可见，不同于十年前网页端的应用，现在浏览器上的应用实现已经非常复杂了。



图 1-2 腾讯课堂桌面浏览器端首页部分用户界面

在这个页面中，浏览器发起的直接网络请求多达 190 个，图片请求约为 160 个，在带有大量高清图片下载的条件下，页面内容体积大小大约只有 2.2MB，平均一个请求大约只有 11.6KB，通过用户操作触发的间接网络请求数可以达到上千个，而在没有使用浏览器缓存的情况下，从用户打开页面网址请求到浏览器页面显示内容却只用了 480~520ms。需要做到这些是一件很不容易的事情，很多方面都需要考虑。首先，网页数据内容多的情况下代码怎么实现？请求数目大时怎么做到快速加载？图片请求数多时内容体积怎样做到尽可能小？网页体积大时怎样在 500ms 内展示给用户？用户操作种类繁杂时怎么去管理实现……然而，我们要明白的是，这个用户首页也只是一个相对比较复杂的页面，还有比这更加复杂且内容更多的网页，那我们又该怎样来解决这些问题呢？

现代前端 Web 应用内容变得更加庞杂，通常也是以多平台的形式展现。仍以腾讯课堂的应用为例，腾讯课堂首页在移动端浏览器下展示的部分用户界面如图 1-3 所示。在移动端，用户打开页面时默认加载的内容大小不到 100KB，页面内容全部加载完成时总请求数约为 50 个，总内容大小约为 200KB，页面中除图片外主要的内容请求不到 10 个，在无缓存的情况下页面

内容平均加载时间约为 1000ms。和桌面端浏览器（这里主要指个人计算机上面安装运行的浏览器）加载的内容相比有些不同，页面请求数量和体积大幅减小，页面内容结构也显得更加简单，但是加载时间却更长了，这又是为什么呢？



图 1-3 腾讯课堂移动浏览器端首页部分用户界面

先不急于回答这些问题，从整体上来看，和互联网应用初始阶段的静态页面相比，现代 Web 浏览器应用具有一些明显的特点，例如页面内容更加复杂、用户交互更多、涉及的平台更广、用户的访问实时性要求更高等。上面的案例只是现代 Web 前端应用的一个缩影，也是一个非常典型的应用案例。要解决以上的这些问题，我们需要先了解一些必要的前端技术知识。

### 1.1.2 现代Web前端技术概述

为了解决上一节中的问题，我们从以下几个方面来思考。

- 页面内容多而复杂，怎样保证开发效率？通常，在前端项目实践中，我们需要借助符合特定场景的前端框架来提高开发效率，例如使用 jQuery、MVVM 等开发框架，对常用的 HTML DOM（Document Object Model，文档对象模型，是指 HTML 内容通过浏

浏览器解析后建立的具有节点父子关系的树形对象)操作进行高效封装,大大简化开发工作量,提高效率。

```
document.getElementById('text').innerHTML = '这是一段文本';
```

在 jQuery 框架里面就可以如下简洁高效地实现。

```
$('#text').html('这是一段文本');
```

- 页面内容多且复杂,项目的管理和维护该如何去做?前端项目代码越来越多,结构越来越复杂,如何实现项目的管理将直接决定后期的维护成本。所以我们必须考虑用模块化和组件化的思路来管理,所谓的模块化和组件化是指采用代码管理中分治的思想,将复杂的代码结构拆分成多个独立、简单、解耦合的结构或文件分开管理,使项目结构更加清晰。例如,某新闻版块组件的内容可以用下面的 HTML 结构来表示。

```
<section class="ui-news">
  <h2>这是新闻标题</h2>
  <article>这是第一篇新闻标题</article>
</section>
```

如果页面的内容增多,我们可以借助 Web Component (HTML5 的一种组件化规范)注册一个<news>的标签来直接引入使用,然后再使用<news>这个标签来代替上面组件的内容进行管理维护,页面中其他的标签内容也可以这样来处理。

```
<news></news>
```

```
// 实际引入组件 news.html 的内容
```

```
<section class="ui-news">
  <h2>这是新闻标题</h2>
  <article>这是第一篇新闻标题</article>
</section>
```

- 页面加载的内容很多,怎样保证尽快将网页内容显示给用户?在页面内容较多、较复杂的情况下,为了让用户尽可能快速看到内容,我们可以通过异步的方式来实现,即将一部分内容先展示给用户,然后根据用户的操作,异步加载用户需要的其他内容,避免用户长时间等待。
- 怎样限制页面内图片的大小以保证页面快速展示?通常,网页上的图片都是比较小的,下载时也比较消耗流量。在上节的这个用户首页中图片请求大约为 160 个,那么我们怎样保证页面下载时消耗的流量尽可能小呢?这可能就需要考虑图片的优化处理了,如使用更高压缩比 webp 格式的图片,在图片质量不降低的情况下,可以大幅度减小图片的网络流量消耗,提高图片加载速度。

```

```

- 对于重复打开的页面，能否让浏览器不再向服务器请求重复的内容呢？其实合理地利用文件缓存就能做到这一点，这样可以大幅度提高网页资源的加载速度，而且幸运的是，浏览器默认可以支持文件缓存，对于一段时间内浏览器的重复请求，服务器可能会返回 HTTP 的 304 状态码或者不发送请求，让浏览器直接从本地读取内容。
- 针对上一节中关于移动端提出的最后一个问题，如果页面地址在移动端浏览器打开，又该怎么处理呢？通常，我们认为移动端和桌面端浏览器相比，有以下几个明显劣势：移动端设备计算资源和网络资源比较有限；移动端设备 CPU 处理速度较慢且网速也相对较低，加载和解析同样的内容需要更长的时间；移动端浏览器受屏幕大小限制，一次能展示的内容有限；移动端设备通常没有键盘和鼠标等外部设备，用户交互的难度增大；移动端浏览器的整体性能不如桌面端浏览器。所以使用移动端浏览器打开一个 Web 页面时，我们常常会考虑让它打开一个用户界面和内容更加简洁的页面。例如，在移动端上访问桌面浏览器端腾讯课堂首页 <https://ke.qq.com/index.html> 将会自动定位到 <https://m.ke.qq.com/index.html> 页面上。尽管这样可以让移动端浏览器加载更少的内容，但所需要的时间还是比较长，仍需要进行更多的优化来提升加载速度。

通过思考上述问题，我们可以大大优化用户访问页面时的体验。与此同时，也涉及了前端开发框架、模块化和组件化开发、资源异步加载、响应式站点开发、缓存和前端优化等多个方面的技术知识。

### 1.1.3 Web前端技术发展

除了从 Web 应用的角度上考虑，我们也可以从前端工程师的角度出发，通过一段代码来看看现代前端的一些特点。作为一个前端工程师，目前我们很可能会这样来写一个 HTML 文件。

```
<!DOCTYPE html>
<html lang="zh-cn" font-size="67.5px">
<head>
  <meta charset="utf-8">
  <title>页面标题</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0,maximum-scale=
1.0,user-scalable=no">
  <meta name="format-detection" content="telephone=no">
  <meta name="keywords" content="页面 SEO 关键字"/>
  <meta name="description" content="页面的描述信息"/>
  <meta itemprop="name" content="页面标题">
  <meta itemprop="image" content="/img/logo-rich.png"/>
  <meta http-equiv="Cache-Control" content="max-age=7200" />
```

```

...
<link rel="dns-prefetch" href="//my.domain.com"/>
<link href="//my.domain.com/main.css" rel="stylesheet">
</head>
<body>
  <header id="hd">
    <video src="./a.m3u8" width="300" height="220"></video>
  </header>
  <section id="bd">
    <picture>
      <source media="(min-width: 375px)" src="med-res.jpg">
      <source media="(min-width: 414px)" src="high-res.jpg">
      
    </picture>
  </section>
  <footer id="ft">
    <embed src="./rect.svg" width="300" height="100" type="image/svg+xml"/>
    <canvas id="canvas"></canvas>
  </footer>
  <script data-main="main" src="mod.js"></script>
</body>
</html>

```

或许你还不理解这里每一行代码的意思，也不了解为什么要这么写。那么，我们先来看一个之前写的页面。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>网站标题</title>
  <meta name="keywords" content="页面 SEO 关键字"/>
  <meta name="description" content="页面的描述"/>
  <meta http-equiv="Cache-Control" content="max-age=7200" />
  <link href="//my.domain.com/main.css" rel="stylesheet">
</head>
<body>
  <div id="hd">
    <embed src="player.swf" width="300" height="220"></embed>
  </div>
  <div id="bd">
    
  </div>
  <div id="ft">
    ...
  </div>
  <script src="main.js"></script>

```

```
</body>
</html>
```

相比这个更早的网页结构，我们现在写的 HTML 结构有些地方变简单了，但是更多的地方变复杂了。在现代浏览器页面开发中，我们通常会使用 HTML5 的新规范，这样能够一定程度上简化开发过程，提高开发效率，也有利于搜索引擎进行检索。当然这些只是基本结构模板的举例，在现在的页面代码中，我们甚至还可能看到如下的使用方法。

```
<link href="./x-image.html" rel="import">
<x-image width="300" height="200"></x-image>
```

这是 HTML5 中组件化 Web Component 的一种实现方式，它通过自定义标签的方式来封装一部分独立的结构功能代码块。另外在 Angular2 框架中，前端页面组件也可以定义如下：使用 TypeScript 语言的装饰器特性来描述声明一个前端页面组件。

```
@Component({ selector: 'my-app',
  template: `<hello-form></hello-form>`,
  directives:[helloFormComponent]
});
```

不仅如此，除了浏览器上的开发，JavaScript 前端脚本语言在 Node.js（Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境，使用了事件驱动、非阻塞式 I/O 的模型，使其轻量又高效，它使用的包管理器为 npm，是目前全球最大的开源库生态系统）服务器端也可以进行自由高效的开发，例如结合 Koa（Koa 是一个简洁高效的 Node.js 端 Web 框架）Web 框架，我们就可以非常便捷地加载中间件创建一个 Web 服务来运行。

```
'use strict';

const http = require('http');
const koa = require('koa');
const serve = require('koa-static');
const koaBody = require('koa-body');
const router = require('./routes');

// 创建一个 Koa 应用
const app = koa();
app.keys = ['site.com'];

// 加载中间件
app.use(serve('./pages')).use(serve('./static'));
app.use(koaBody({
  formidable: {
    uploadDir: __dirname
  }
}));
```



```
app.use(router.routes());

// 创建服务器监听
http.createServer(app.callback()).listen(8000);
console.log('Server listening on port 8000');
```

这些示例代码中涉及了一些 ECMAScript 6+标准的新特性,在 Web 前端开发实践中,这些新的特性早已经深入到项目开发的各个环节当中了,ECMAScript 6+和原来的语言规范相比更加简洁、高效。我们知道,前端技术栈中的 JavaScript 语言不仅能在浏览器端解析,也能在 Node.js 服务器上运行,而且目前已经被广泛使用在各类企业后端服务中。但是这里要说的是,前端技术上的这些变化仅仅发生在四年之间,而且这里列举的只是前端技术变化中极少的一部分。可以说互联网和移动互联网的诞生与井喷式的发展促进了前端技术的发展,带来了这些巨大的变化。我们不能确定未来前端技术会变成什么样子,但可以肯定的是,从现在起,四年后又将是完全不同的。

我们必须承认,前端工程师需要做的事情远不只是将一个设计稿图片在页面上实现这么简单,仅从 Web 前端结构的开发实现模式上来看就已经发生了翻天覆地的变化。

如图 1-4 所示,前端结构的开发实现模式先后经历了静态黄页时期、服务器组装动态网页数据时期、后端为主的 MVC 模式时期、前后端分离时期、纯前端 MV\*为主和中间层直出时期、前端 Virtual DOM、MNV\*、前后端同构时期。

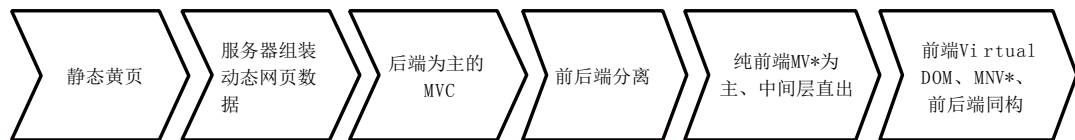


图 1-4 前端应用开发模式演变

Web 前端的概念随着互联网的出现而产生,Web 页面结构从最早的静态黄页形式,渐渐变成使用后端数据组装输出动态页面的形式。数据复杂后,出现了以后端语言为核心的 MVC 开发模式。为了方便管理更复杂的 Web 项目,有人提出了前后端分离开发管理的方案,让前端工程师能够专注于浏览器端的交互逻辑实现。前后端分离方式出现后,前端快速发展并进入了以纯前端 MV\*框架为主的时代,现在前端则可以利用更加灵活且与组件化结合的 Virtual DOM 交互模式来实现,或者选择前后端同构、跨终端 MNV\*的开发模式来应对不同的业务场景。

由此说明,Web 前端技术的发展非常迅速而且已经发生了巨大的变化,但通过这一系列的变化我们可以看出,前端从无到有且发展到现在的不变宗旨是,一直持续在以效率和质量为最

终导向的道路上探索前进，并且未来关于 Web 技术效率和质量这两方面的探索仍会有增无减。效率方面，从前后端分离到出现各种封装的前端框架，都在解决一个前端编程开发效率的问题。前端性能作为前端质量的一个重要部分一直倍受关注，而现在前端 Virtual DOM 和 MNV\*交互模式等的实现思路，就是为解决前端交互性能问题而出现的。当然这仅仅是在前端开发框架上做出的完善和改进，此外相关前端工程、自动化构建、组件化、前端优化等技术解决方案的出现，也为现代前端开发的效率和质量提升做出了重要的贡献。

## 1.2 浏览器应用基础

### 1.2.1 浏览器组成结构

在介绍浏览器组成结构之前，我们先来看一个以前经常被问到的问题：从我们打开浏览器输入一个网址到页面展示网页内容的这段时间内，浏览器和服务端都发生了什么事情？我们直接来看一个相对简洁但比较清晰的过程描述。

- 在接收到用户输入的网址后，浏览器会开启一个线程来处理这个请求，对用户输入的 URL 地址进行分析判断，如果是 HTTP 协议就按照 HTTP 方式来处理。
- 调用浏览器引擎中的对应方法，比如 WebView 中的 loadUrl 方法，分析并加载这个 URL 地址。
- 通过 DNS 解析获取该网站地址对应的 IP 地址，查询完成后连同浏览器的 Cookie、userAgent 等信息向网站目的 IP 发出 GET 请求。
- 进行 HTTP 协议会话，浏览器客户端向 Web 服务器发送报文。
- 进入网站后台上的 Web 服务器处理请求，如 Apache、Tomcat、Node.js 等服务器。
- 进入部署好的后端应用，如 PHP、Java、JavaScript、Python 等后端程序，找到对应的请求处理逻辑，这期间可能会读取服务器缓存或查询数据库等。
- 服务器处理请求并返回响应报文，此时如果浏览器访问过该页面，缓存上有对应资源，会与服务器最后修改记录对比，一致则返回 304，否则返回 200 和对应的内容。
- 浏览器开始下载 HTML 文档（响应报头状态码为 200 时）或者从本地缓存读取文件内容（浏览器缓存有效或响应报头状态码为 304 时）。

- 浏览器根据下载接收到的 HTML 文件解析结构建立 DOM (Document Object Model, 文档对象模型) 文档树, 并根据 HTML 中的标记请求下载指定的 MIME 类型文件 (如 CSS、JavaScript 脚本等), 同时设置缓存等内容。
- 页面开始解析渲染 DOM, CSS 根据规则解析并结合 DOM 文档树进行网页内容布局和绘制渲染, JavaScript 根据 DOM API 操作 DOM, 并读取浏览器缓存、执行事件绑定等, 页面整个展示过程完成。

整个过程中使用到了较多的浏览器功能, 如用户地址栏输入框、网络请求、浏览器文档解析、渲染引擎、JavaScript 执行引擎、客户端存储等。下面, 我们具体来看一下浏览器的主要结构。

如图 1-5 所示, 通常我们认为浏览器主要由七部分组成: 用户界面、网络、JavaScript 引擎、渲染引擎、UI 后端、JavaScript 解释器和持久化数据存储。用户界面包括浏览器中可见的地址输入框、浏览器前进返回按钮、打开书签、打开历史记录等用户可操作的功能选项。浏览器引擎可以在用户界面和渲染引擎之间传送指令或在客户端本地缓存中读写数据等, 是浏览器中各个部分之间相互通信的核心。浏览器渲染引擎的功能是解析 DOM 文档和 CSS 规则并将内容排版到浏览器中显示有样式的界面, 也有人称之为排版引擎, 我们常说的浏览器内核主要指的就是渲染引擎。网络功能模块则是浏览器开启网络线程发送请求或下载资源文件的模块, 例如 DOM 树解析过程中请求静态资源首先是通过浏览器中的网络模块发起的, UI 后端则用于绘制基本的浏览器窗口内控件, 比如组合选择框、按钮、输入框等。JavaScript 解释器则是浏览器解释和执行 JavaScript 脚本的部分, 例如 V8 引擎。浏览器数据持久化存储则涉及 cookie、localStorage 等一些客户端存储技术, 可以通过浏览器引擎提供的 API 进行调用。

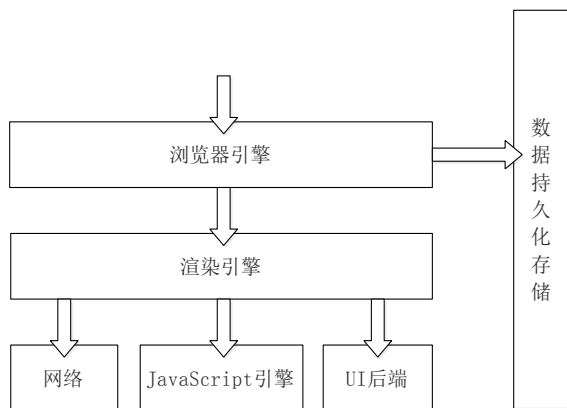


图 1-5 浏览器组成结构

作为前端开发者，我们需要重点理解渲染引擎的工作原理，灵活运用数据持久化存储技术，因为实际的开发工作中这部分的操作比较多，而其他几个部分都是由浏览器决定的，开发者能控制的部分相对较少。接下来我们就重点讲解这两个部分。

目前，用户使用的主流浏览器内核有 4 类：Trident 内核，例如 Internet Explorer、360 浏览器、搜狗浏览器等；Gecko 内核，Netscape 6 及以上版本，还有 Firefox、SeaMonkey 等；Presto 内核，主要是 Opera 7 及以上还在使用（Opera 内核原来为 Presto，现使用的是 Blink 内核）；Webkit 内核，主要是 Safari、Chrome 浏览器在使用的内核，也是特性上表现较好的浏览器内核。另外，Blink 内核其实是 WebKit 的一个分支，添加了一些优化新特性，例如跨进程的 iframe，将 DOM 移入 JavaScript 中来提高 JavaScript 对 DOM 的访问速度等，目前较多的移动端应用内嵌的浏览器内核也渐渐开始采用 Blink。

## 1.2.2 浏览器渲染引擎简介

### 渲染引擎的主要工作流程

渲染引擎在浏览器中主要用于解析 HTML 文档和 CSS 文档，然后将 CSS 规则应用到 HTML 标签元素上，并将 HTML 渲染到浏览器窗口中以显示具体的 DOM 内容。如图 1-6 所示，浏览器通过网络模块下载 HTML 文件后进行页面解析渲染，流程主要包括以下几个步骤：解析 HTML 构建 DOM 树、构建渲染树、渲染树布局、绘制渲染树。

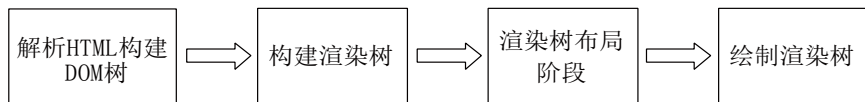


图 1-6 渲染引擎工作流程

解析 HTML 构建 DOM 树时渲染引擎会先将 HTML 元素标签解析成由多个 DOM 元素对象节点组成的且具有节点父子关系的 DOM 树结构，然后根据 DOM 树结构的每个节点顺序提取计算使用的 CSS 规则并重新计算 DOM 树结构的样式数据，生成一个带样式描述的 DOM 渲染树对象。DOM 渲染树生成结束后，进入渲染树的布局阶段，即根据每个渲染树节点在页面中的大小和位置，将节点固定到页面的对应位置上，这个阶段主要是元素的布局属性（例如 position、float、margin 等属性）生效，即在浏览器中绘制页面上元素节点的位置。接下来就是绘制阶段，将渲染树节点的背景、颜色、文本等样式信息应用到每个节点上，这个阶段主要是元素的内部显示样式（例如 color、background、text-shadow 等属性）生效，最终完成整个 DOM 在页面上的绘制显示。

这里我们要关注的是渲染树的布局阶段和绘制阶段。页面生成后,如果页面元素位置发生变化,就要从布局阶段开始重新渲染,也就是页面重排,所以页面重排一定会进行后续重绘;如果页面元素只是显示样式改变而布局不变,那么页面内容改变将从绘制阶段开始,也称为页面重绘。重排通常会导致页面元素几何大小位置发生变化且伴随着重新渲染的巨大代价,因此我们要尽可能避免页面的重排,并减少页面元素的重绘。

渲染引擎对 DOM 渲染树的解析和输出是逐行进行的,所以渲染树前面的内容可以先渲染展示,这样就保证了较好的用户体验。另外也尽量不要在 HTML 显示内容中插入 script 脚本等标签,script 标签内容的解释执行常常会阻塞页面结构的渲染。

通常情况下,不同浏览器内核的解析渲染过程也略有不同,我们以 Chrome、Safari 浏览器的 Webkit 内核和 Firefox 浏览器的 Gecko 内核为例,来看看渲染引擎工作流程的这四个步骤具体是怎样完成的。

图 1-7 和图 1-8 分别为 Webkit 内核和 Gecko 内核渲染 DOM 的主要流程。可以看出,两种渲染引擎工作流程的主要区别在于解析 HTML 或 CSS 文档生成渲染树的过程: Webkit 内核中的 HTML 和 CSS 解析可以认为是并行的;而 Gecko 则是先解析 HTML,生成内容 Sink (Content Sink 可以认为是构建 DOM 结构树的工厂方法) 后再开始解析 CSS。这两种渲染引擎工作过程中使用的描述术语也不一样: Webkit 内核解析后的渲染对象被称为渲染树 (Render Tree), 而 Gecko 内核解析后的渲染对象则称为 Frame 树 (Frame Tree)。但是它们主要的流程是相似的,都经过 HTML DOM 解析、CSS 样式解析、渲染树生成和渲染树绘制显示阶段。一般渲染引擎的解析过程中都包含了 HTML 解析和 CSS 解析阶段,这也是渲染引擎解析流程中最重要的两个部分,接下来就看看 HTML 文档解析和 CSS 规则解析具体是怎样进行的。

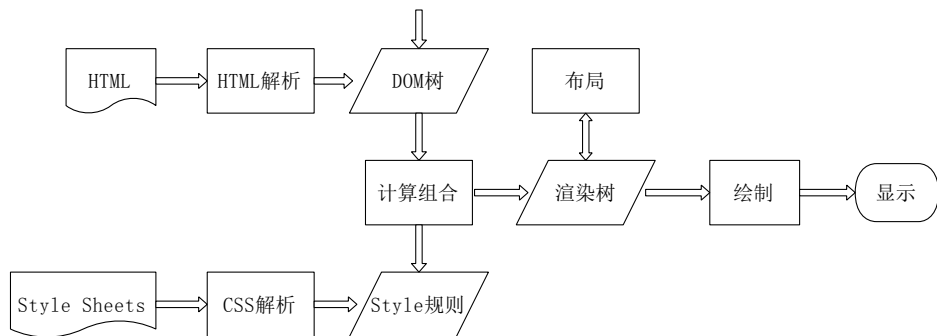


图 1-7 Webkit 内核渲染 DOM 流程

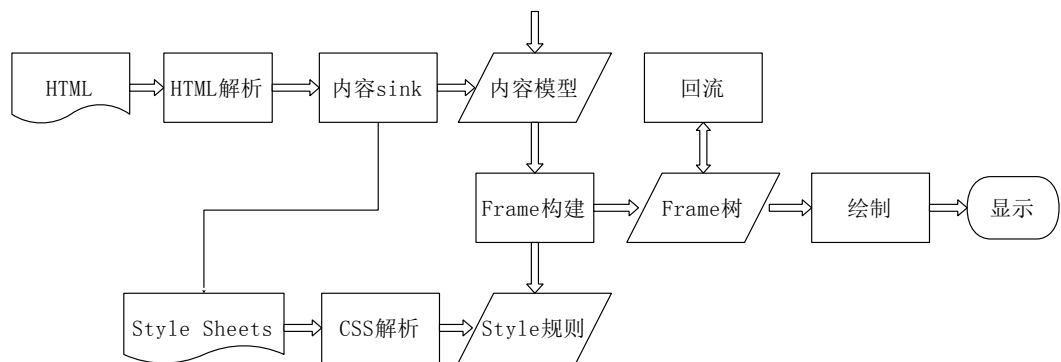


图 1-8 Gecko 内核渲染 DOM 流程

👉 HTML文档解析

HTML 文档解析过程是将 HTML 文本字符串逐行解析生成具有父子关系的 DOM 节点树对象的过程，例如下面的 HTML 结构，通过解析 HTML 文档就生成了如图 1-9 所示的由多个不同 DOM 元素对象组成的 DOM 树。

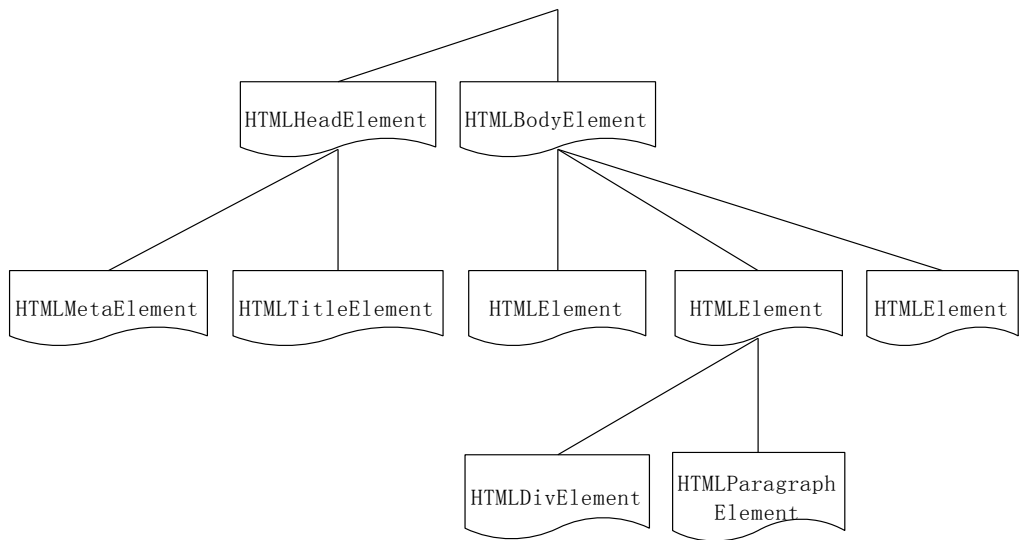


图 1-9 DOM 结构解析示意图

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>页面标题</title>
</head>
<body>
  <header>头部内容</header>
  <section>
    <div>块元素</div>
    <p>段落</p>
  </section>
  <footer>底部</footer>
</body>
</html>

```

渲染引擎在 HTML 解析阶段会对 HTML 文本的标签进行分析,同时创建 DOM 对象,最终生成图 1-9 所示的 DOM 对象树。需要注意的是, <header>、<nav>、<section>、<footer> 这些布局类标签的 DOM 类型均为 HTML 元素,此外 HTML 中不同标签的每个元素对应的原始类型也可以是不一样的。

```

let element = document.getElementById('div');
let type = Object.prototype.toString.call(element).slice(8, -1);
console.log(type); // HTMLDivElement

```

我们通过这一段代码就可以获取不同 DOM 元素对应的原始类型了,和 HTML 标准一样,DOM 元素的原始类型标准也是由 W3C 组织指定的,具体更多常用的类型可以参考下面的内容。

### ○ 根元素

```
<html> (HTMLHtmlElement)
```

### ○ 文件数据元素

```

<head> (HTMLHeadElement)
<title> (HTMLTitleElement)
<base> (HTMLBaseElement)
<link> (HTMLLinkElement)
<meta> (HTMLMetaElement)
<style> (HTMLStyleElement)
<script> (HTMLScriptElement)
<noscript> (HTMLNoScriptElement)

```

### ○ 文件区域元素

```
<body> (HTMLBodyElement)
```

```
<section> (HTMLElement)
<nav> (HTMLElement)
<article> (HTMLElement)
<aside> (HTMLElement)
<h1> <h2> <h3> <h4> <h5> <h6> (HTMLHeadingElement)
<hgroup> (HTMLElement)
<header> (HTMLElement)
<footer> (HTMLElement)
<address> (HTMLElement)
```

### ○ 群组元素

```
<p> (HTMLParagraphElement)
<hr> (HTMLHRElement)
<pre> (HTMLPreElement)
<blockquote> (HTMLQuoteElement)
<ol> (HTMLOListElement)
<ul> (HTMLUListElement)
<li> (HTMLLIElement)
<dl> (HTMLDListElement)
<dt> (HTMLElement)
<dd> (HTMLElement)
<div> (HTMLDivElement)
```

### ○ 文字层级元素

```
<a> (HTMLAnchorElement)
<em> (HTMLElement)
<strong> (HTMLElement)
<small> (HTMLElement)
<i> (HTMLElement)
<b> (HTMLElement)
<span> (HTMLSpanElement)
<br> (HTMLBRElement)
```

### ○ 编修记录元素

```
<ins> (HTMLModElement)
<del> (HTMLModElement)
```

### ○ 内嵌媒体元素

```
<img> (HTMLImageElement)
```



```
<iframe> (HTMLIFrameElement)
<embed> (HTMLEmbedElement)
<object> (HTMLObjectElement)
<param> (HTMLParamElement)
<video> (HTMLVideoElement)
<audio> (HTMLAudioElement)
<source> (HTMLSourceElement)
<canvas> (HTMLCanvasElement)
```

### ○ 表格元素

```
<table> (HTMLTableElement)
<caption> (HTMLTableCaptionElement)
<tbody> (HTMLTableSectionElement)
<thead> (HTMLTableSectionElement)
<tfoot> (HTMLTableSectionElement)
<tr> (HTMLTableRowElement)
<td> (HTMLTableDataCellElement)
<th> (HTMLTableHeaderCellElement)
```

### ○ 窗体元素

```
<form> (HTMLFormElement)
<fieldset> (HTMLFieldSetElement)
<legend> (HTMLLegendElement)
<label> (HTMLLabelElement)
<input> (HTMLInputElement)
<button> (HTMLButtonElement)
<select> (HTMLSelectElement)
<datalist> (HTMLDataListElement)
<option> (HTMLOptionElement)
<textarea> (HTMLTextAreaElement)
```

### ○ 交互式元素

```
<details> (HTMLDetailsElement)
<summary> (HTMLElement)
<command> (HTMLCommandElement)
<menu> (HTMLMenuElement)
```

渲染引擎通过解析 HTML 文本形成了对象化的 DOM 树，但要将 DOM 树渲染到浏览器窗口中形成有样式的内容，仍需要结合 CSS 规则生成一个带有节点 CSS 样式描述的 DOM 树。

DOM 元素标签和 DOM 元素对象虽然都是用来描述 DOM 结构的，但 DOM 元素标签是指文本化的 HTML 标识，而 DOM 元素对象则是指经过渲染引擎 DOM 解析后生成的具有节点父子关系的树形对象。

## 👉 CSS解析

CSS 解析和 HTML 解析类似，首先也要通过词法解析生成 CSS 分析树，而且也使用特定的 CSS 文本语法来实现，不同的是，HTML 是使用类似 XML 结构的语法解析方式来完成分析的。以下面的解析 CSS 代码为例。

```
html, body{
    margin: 0;
    color: red;
}
header, section, footer, div, p{
    margin-top: 10px;
}
```

通过分析 CSS 文档，会生成如图 1-10 所示的 CSS 规则树状图，CSSRule 会保持每个不同元素和对应样式的映射关系。在渲染树逐行生成的阶段，DOM 树中的节点会在 CSS 分析树中根据元素、类、id 选择器来提取与之对应元素的一条或多条 CSSRule，进行 CSS 规则的层叠和权重计算，得到最终生效的样式 CSSRule 并添加到 DOM 渲染树上，当每个 DOM 节点提取 CSS 样式完成时，用于页面布局和绘制的 DOM 渲染树便形成了。在一个已经形成的 DOM 渲染树中，节点的 CSS 规则可通过 `document.defaultView.getComputedStyle(element, null)` 方法来获取查看。图 1-11 所示为上面文档中 `<header>` 元素标签通过浏览器渲染后，最终计算得到的 `margin` 值内容。

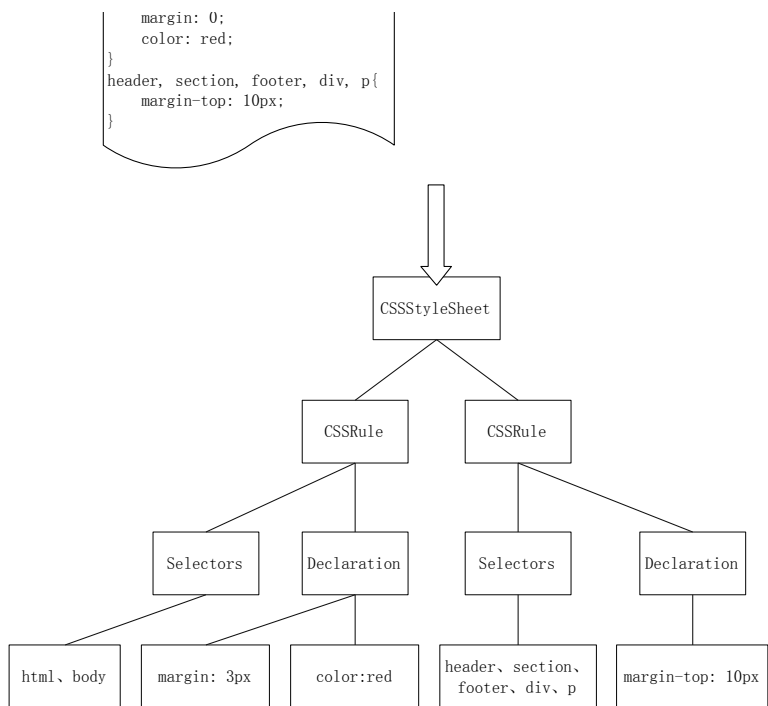


图 1-10 CSS 结构解析示意图

```
justifyContent: "normal"
left: "auto"
length: 258
letterSpacing: "normal"
lightingColor: "rgb(255, 255, 255)"
lineHeight: "normal"
listStyle: "disc outside none"
listStyleImage: "none"
listStylePosition: "outside"
listStyleType: "disc"
margin: "10px 0px 0px"
marginBottom: "0px"
marginLeft: "0px"
marginRight: "0px"
marginTop: "10px"
marker: ""
markerEnd: "none"
markerMid: "none"
markerStart: "none"
mask: "none"
maskType: "luminance"
maxHeight: "none"
maxWidth: "none"
maxZoom: ""
minHeight: "0px"
minWidth: "0px"
```

图 1-11 解析完成最终样式

一个节点上多条不同的样式规则是通过权重的方式来计算的。关于 CSS 规则的权重计算，一般认为是 !important > 内联样式规则（权重 1000）> id 选择器（权重 100）> 类选择器（权重 10）> 元素选择器（权重 1）。在渲染树生成阶段，当多个 CSSRule 对应到同一个元素节点中时，这种权重计算方式的作用就体现出来了，一般只有权重最高的样式规则才会生效。

参考资料：<http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/>。

### 1.2.3 浏览器数据持久化存储技术

这里所说的数据持久化存储主要是针对浏览器的，所以我们统称为浏览器缓存（Browser Caching），浏览器缓存是浏览器端用于在本地保存数据并进行快速读取以避免重复资源请求的传输机制的统称。有效的缓存可以避免重复的网络资源请求并让浏览器快速地响应用户操作，提高页面内容的加载速度。浏览器端缓存的实现机制种类较多，一般可以分为九种，如图 1-12 所示。打开 Chrome 浏览器的调试模式，Application 左侧就列举了现代浏览器的 8 种缓存机制：HTTP 文件缓存、LocalStorage、SessionStorage、indexedDB、Web SQL、Cookie、CacheStorage、Application Cache，另外还有一种使用不太多的 Flash 缓存方式。下面逐个介绍这 9 种缓存机制的原理和使用场景。

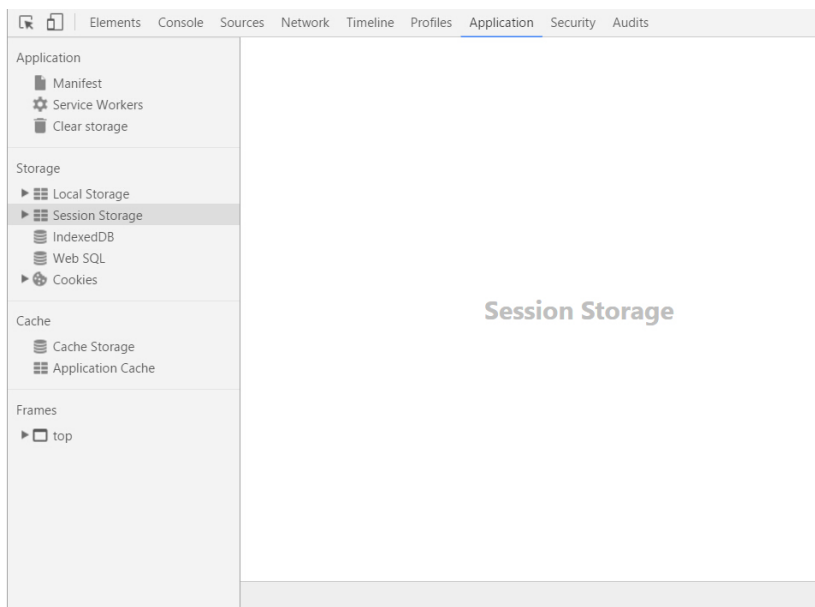


图 1-12 浏览器缓存管理界面

## HTTP文件缓存

HTTP 文件缓存是基于 HTTP 协议的浏览器端文件级缓存机制。在文件重复请求的情况下，浏览器可以根据 HTTP 响应的协议头信息判断是从服务器端请求文件还是从本地读取文件，Chrome 控制台下的 Frames 就可以查看浏览器的 HTTP 文件资源缓存列表内容。

图 1-13 是浏览器 HTTP 文件缓存判断机制的流程图。针对浏览器的重复 HTTP 请求，浏览器和服务器判断是否使用浏览器端文件缓存的过程如下。

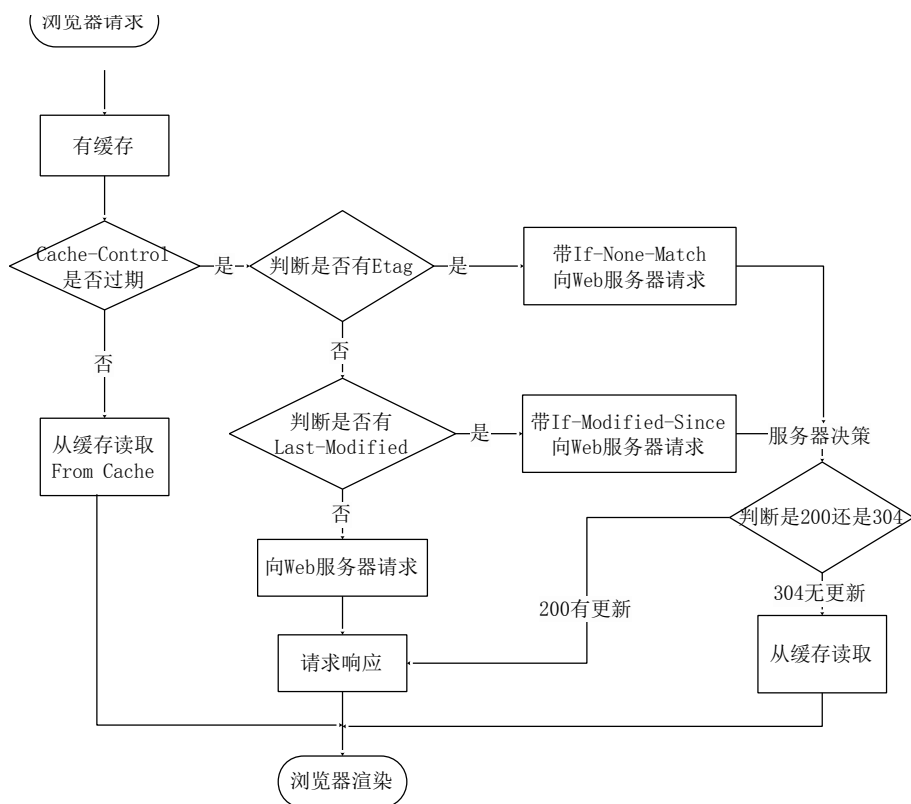


图 1-13 HTTP 文件缓存判断机制流程

1. 浏览器会先查询 Cache-Control（这里用 Expires 判断也是可以的，但是 Expires 一般设置的是绝对过期时间，Cache-Control 设置的是相对过期时间）来判断内容是否过期，如果未过期，则直接读取浏览器端缓存文件，不发送 HTTP 请求，否则进入下一步。

2. 在浏览器端判断上次文件返回头中是否含有 Etag 信息，有则连同 If-None-Match 一起向服务器发送请求，服务端判断 Etag 未修改则返回状态 304，修改则返回 200，否则进入下一步。

3. 在浏览器端判断上次文件返回头中是否含有 Last-Modified 信息，有则连同 If-Modified-Since 一起向服务器发送请求，服务端判断 Last-Modified 是否失效，失效则返回 200，未失效则返回 304。

4. 如果 Etag 和 Last-Modified 都不存在，直接向服务器请求内容。

HTTP 缓存可以在文件缓存生效的情况下让浏览器从本地读取文件，不仅加快了页面资源加载，同时节省网络流量，所以在 Web 站点配置中要尽可能利用缓存来优化请求过程。在 HTML 中，我们可以添加<meta>中的 Expires 或 Cache-Control 来设置，需要注意的是，一般这里 Cache-Control 设置 max-age 的时间单位是秒，如果同时设置了 Expires 和 Cache-Control，则只有 Cache-Control 的设置生效。

```
<meta http-equiv="Expires" content="Mon, 20 Jul 2016 23:00:00 GMT" />
<meta http-equiv="Cache-Control" content="max-age=7200" />
```

当然服务端也需要进行对应设置，Node.js 服务器可以使用中间件来这样设置静态资源文件的缓存时间，例如我们可以结合 Koa Web 框架和 koa-static 中间件如下设置实现。

```
const static= require('koa-static');
const app = koa();
app.use(static ('./pages', {
  maxage: 7200
}));
```

## 📁 localStorage

localStorage 是 HTML5 的一种本地缓存方案，目前主要用于浏览器端保存体积较大的数据（如 AJAX 返回结果等），需要了解的是，localStorage 在不同浏览器中有长度限制且各不相同。

```
// localStorage 核心 API:
localStorage.setItem(key, value) //设置 localStorage 存储记录
localStorage.getItem(key)        //获取 localStorage 存储记录
localStorage.removeItem(key)      //删除该域下单条 localStorage 存储记录
localStorage.clear()              //删除该域名下所有 localStorage 存储记录
```

如表 1-1 所示，localStorage 基本支持目前的主流浏览器，在 Internet Explorer 8 以上最大限制为 5MB，在 Chrome 或 Safari 浏览器里面的大小限制约为 2.6MB。另外 localStorage 常用的 API 也较少，使用起来极其方便，核心方法只有 setItem()、getItem()、removeItem()、clear()。值得注意的是，这里的大小限制指的是单个域名下 localStorage 的大小，所以 localStorage 中不适合存放过多的数据，如果数据存放超过最大限制可能会读取报错，因此在使用之后最好移除不再使用的数据。

表 1-1 localStorage 浏览器支持与大小限制

浏览器	最大长度
Internet Explorer 8 以上	5MB
Firefox 8 以上	5.24MB
Opera	2MB
Chrome、Safari	2.6MB

此外，localStorage 只支持简单数据类型的读取，为了方便 localStorage 读取对象等格式的内容，通常需要进行一层安全封装再引入使用。

```
let rkey = /^[0-9A-Za-z_@-]*$/;
let store;

// 转换对象
function init() {
  if (typeof store === 'undefined') {
    store = window['localStorage'];
  }
  return true;
}

// 判断 localStorage 的 key 值是否合法
function isValidKey(key) {
  if (typeof key !== 'string') {
    return false;
  }
  return rkey.test(key);
}

exports = {
  // 设置 localStorage 单条记录
  set (key, value) {
    let success = false;
    if (isValidKey(key) && init()) {
      try {
        value += '';
        store.setItem(key, value);
        success = true;
      } catch (e) {}
    }
    return success;
  },

  // 读取 localStorage 单条记录
  get (key) {
```

```
    if (isValidKey(key) && init()) {
      try {
        return store.getItem(key);
      } catch (e) {}
    }
    return null;
  },

  // 移除 localStorage 单条记录
  remove (key) {
    if (isValidKey(key) && init()) {
      try {
        store.removeItem(key);
        return true;
      } catch (e) {}
    }
    return false;
  },

  // 清除 localStorage 所有记录
  clear () {
    if (init()) {
      try {
        for (let key in store) {
          store.removeItem(key);
        }
        return true;
      } catch (e) {}
    }
    return false;
  }
};

module.exports = exports;
```

尽管单个域名下 `localStorage` 的大小是有限制的，但是可以用 `iframe` 的方式使用多个域名来突破单个页面下 `localStorage` 存储数据的最大限制。另外使用浏览器多个标签页打开同一个域名页面时，`localStorage` 内容一般是共享的。

## 👉 sessionStorage

`sessionStorage` 和 `localStorage` 的功能类似，但是 `sessionStorage` 在浏览器关闭时会自动清空。`sessionStorage` 的 API 和 `localStorage` 完全相同。由于不能进行客户端的持久化数据存储，实际项目中 `sessionStorage` 的使用场景相对较少。



📁 Cookie

Cookie（或 Cookies），指网站为了辨别用户身份或 Session 跟踪而储存在用户浏览器端的数据。Cookie 信息一般会通过 HTTP 请求发送到服务器端。一条 Cookie 记录主要由键、值、域、过期时间和大小组成，一般用于保存用户的网站认证信息。浏览器中 Cookie 的最大长度和单个域名支持的 Cookie 个数由浏览器的不同来决定。如表 1-2 所示，在 Internet Explorer 7 以上版本、Firefox 等浏览器中，支持的 Cookie 记录最多是 50 条，Chrome、Safari 浏览器上则没有限制，Opera 浏览器上最多支持 30 条，而且我们通常认为 Cookie 的最大长度限制为 4KB（4095 字节到 4097 字节）。

表 1-2 Cookie 浏览器支持与大小

浏 览 器	支持域名个数	最大长度
Internet Explorer 7 以上	50 个	4095B
Firefox	50 个	4097B
Opera	30 个	4096B
Chrome、Safari	无限制	4097B

和 localStorage 类似，不同域名之间的 Cookie 信息也是独立的，如果需要设置共享，则可以在被共享域名的服务器端设置 Cookie 的 path 和 domain 来实现，例如 Koa Web 框架下就可以用如下方法来设置 cookie 在相同父域的多个子域中共享，其他的 Web 后端框架也有类似的设置方法。

```
this.cookies.set('username', 'ouven', {
  domain: '.domain.com',
  path: '/'
});
```

浏览器端也可以通过 document.cookie 来获取 Cookie，并通过 JavaScript 来处理解析。但是需要注意的是，Cookie 分为两种：Session Cookie 和持久型 Cookie。Session Cookie 一般不设置过期时间，表示该 Cookie 的生命周期为浏览器会话期间，只要关闭浏览器窗口，Cookie 就会消失，而且 Session Cookie 一般不保存在硬盘上而是保存在内存里；持久型 Cookie 一般会设置过期时间，而且浏览器会将持久型 Cookie 的信息保存到硬盘上，关闭后再次打开浏览器，这些 Cookie 依然有效，直到超过设定的过期时间或被清空才失效。Cookie 设置中有个 HttpOnly 参数，前端浏览器使用 document.cookie 是读取不到 HttpOnly 类型 Cookie 的，被设置为 HttpOnly 的 Cookie 记录只能通过 HTTP 请求头发送到服务器端进行读写操作，这样就避免了服务器端的 Cookie 记录被前端 JavaScript 修改，保证了服务端验证 Cookie 的安全性。

JavaScript 在浏览器端可以通过 `document.cookie` 来读取非 `HttpOnly` 类型的 Cookie 记录, `document.cookie` 的内容通常是下面这种用等号和分号分割的键值对形式的字符串。

```
"Hm_lvt_6225b4f9f1912feb003dd0be6d643a73=1472800594,1473130193;Hm_lpv_6225b4f9f1912feb003dd0be6d643a73=1473130201"
```

在前端浏览器中,如果要对 `document.cookie` 进行修改就比较麻烦了,不过我们同样可以进行统一的封装来达到方便读取和操作 Cookie 的目的。

```
exports = {

  // 获取单条 cookie 记录
  get (n){
    let m = document.cookie.match(new RegExp( "(^| )" + n + "=( [^;]* ) (; | $) " ));
    return !m ? "" : decodeURIComponent( m[2] );
  },

  // 设置单条 cookie 记录
  set (name, value, domain, path, hour){
    let expire = new Date();
    expire.setTime(expire.getTime() + (hour?3600000 * hour:30*24*60*60*1000));
    document.cookie = name + "=" + value + ";" + "expires=" + expire.toGMTString() + ";
    path=" + (path ? path : "/" ) + ";" + (domain ? ("domain=" + domain + ";" ) : "");
  },

  // 删除单条 cookie 记录
  del (name, domain, path) {
    document.cookie = name + "=; expires=Mon, 26 Jul 1997 05:00:00 GMT; path=" + (path ?
    path : "/" ) + ";" + (domain ? ("domain=" + domain + ";" ) : "");
  },

  // 清除 document.cookie
  clear () {
    let rs = document.cookie.match(new RegExp( "( [^;]* ) (?= ( [^;]* ) (; | $) )", "gi" ));
    // 删除所有 cookie
    for (let i in rs){
      document.cookie = rs[i] + "=; expires=Mon, 26 Jul 1997 05:00:00 GMT; path=/; ";
    }
  }
};

module.exports = exports;
```

存储型 Cookie 是存储在浏览器客户端计算机硬盘上的。以 Windows 系统的 Chrome 浏览器为例,浏览器域名的 Cookie 文件存储在 `\documents and settings\userName\cookie\` 文件夹下的 txt 文本文件里,通常文本内容格式为通过一定规则加密的 Cookie 内容,我们打开文件就可以直接查看。

## 📁 WebSQL

WebSQL 是浏览器端用于存储较大量数据的缓存机制，不过这只有较新版本的 Chrome 浏览器支持该机制，并以一个独立浏览器端数据存储规范的形式出现。WebSQL 主要有以下几个特点。

1. WebSQL 数据库 API 实际上不是 HTML5 规范的组成部分，目前只是一种特定的浏览器特性，而且 WebSQL 在 HTML5 之前就已经存在，是单独的规范。
2. WebSQL 将数据以数据库二维表的形式存储在客户端，可以根据需要使用 JavaScript 去读取。
3. WebSQL 与其他存储方式的区别：localStorage 和 Cookie 以键值对的形式存在，WebSQL 为了更便于检索，允许 SQL 语句的查询。
4. WebSQL 可以让浏览器实现小型数据库存储功能，而且使用的数据库是集成在浏览器里面的。

WebSQL API 主要包含三个核心方法：openDatabase()、transaction() 和 executeSql()。例如在 mydatabase 数据库中创建表 t1，并插入两条记录，实现如下。

```
// openDatabase()方法可以打开已经存在的数据库，不存在则创建
let db = openDatabase('mydatabase', '1.0', 'test table', 2 * 1024 * 1024);
let name = [2, 'ouven'];
db.transaction(function (table) {
    table.executeSql('CREATE TABLE IF NOT EXISTS t1 (id unique, msg)');
    table.executeSql('INSERT INTO t1 (id, msg) VALUES (1, "hello")');
    table.executeSql('INSERT INTO t1 (id, msg) VALUES (?, ?)', name);
});

transaction(); // transaction()这个方法允许我们根据情况控制执行事务提交或回滚
executeSql();  // executeSql()用于执行真实的 SQL 查询语句
```

openDatabase() 方法可以打开已存在的数据库，并默认创建不存在的数据库。openDatabase() 中的五个参数分别为数据库名、版本号、描述、数据库大小、创建回调，即使创建回调为 null 也可以创建数据库，transaction() 方法允许我们根据情况控制执行事务提交或回滚，executeSql() 则用于执行真实的 SQL 查询语句。

如果需要读操作，而且是读取已经存在的记录，我们也可以使用一个回调函数来处理查询的结果，实现如下。

```
let db = openDatabase('mydatabase', '2.0', 'test table', 2*1024);
```

```
db.transaction(function (table) {  
  table.executeSql('SELECT * FROM t1', [], function (table, results) {  
    let len = results.rows.length, i;  
    for (i = 0; i < len; i++){  
      console.log(results.rows.item(i).msg);  
    }  
  }, null);  
});
```

由于兼容性问题，加上使用场景比较有限，目前 WebSQL 的应用并不是很广泛，但是作为浏览器缓存技术的一种方式，我们有必要了解它的特点和实现方法。

### 📁 IndexedDB

IndexedDB 也是一个可在客户端存储大量结构化数据并且能在这些数据上使用索引进行高性能检索的一套 API。由于 WebSQL 不是 HTML5 规范，不支持 Internet Explorer 10、Chrome 12 及 Firefox 5 以上版本的浏览器，所以一般推荐使用 IndexedDB 来进行大量数据的存储，其基本实现和 WebSQL 类似，只是使用的 API 规范不一样，WebSQL 使用类似 NoSQL（Not Only SQL，非关系型数据库）数据库的设计实现，读取效率更高。浏览器对 IndexedDB 的大小限制通常约为 50MB，这样就可以将大量的应用数据保存到本地，在本地满足需要搜索的场景。

```
if (database) {  
  database.transaction(function(tx) {  
    tx.executeSql("CREATE TABLE IF NOT EXISTS test (id REAL UNIQUE, text TEXT)", []);  
  });  
}
```

和 WebSQL 类似，目前使用 IndexedDB 的实际应用场景也不是很多，而且将大量数据保存到本地也会造成数据泄露，所以我们了解即可，无须在实际项目中使用。

### 📁 Application Cache

Application Cache 是一种允许浏览器通过 manifest 配置文件在本地有选择性地存储 JavaScript、CSS、图片等静态资源的文件级缓存机制。当页面不是首次打开时，通过一个特定的 manifest 文件配置描述来选择读取本地 Application Cache 里面的文件。所以使用 Application Cache 来实现浏览器应用具有以下三个优势。

1. 离线浏览。通过 manifest 配置描述来读取本地文件，用户可在离线时浏览完整的页面内容。
2. 快速加载。由于缓存资源为本地资源，因此页面加载速度较快。
3. 服务器负载小。只有在文件资源更新时，浏览器才会从服务器端下载，这样就减小了服

务器资源请求的压力。

图 1-14 为 Application Cache 访问页面资源和更新缓存的具体流程。Application Cache 在页面第二次被访问时开始生效。页面打开时优先从 Application Cache 中访问资源，读取资源加载后同时会去请求检查服务端的 manifest 文件是否已更新，如果没有更新，则整个访问过程结束；否则浏览器会去检查 manifest 列表，将更新的内容重新拉取到 Application Cache 中，这样页面第三次被访问时就可以加载到更新后的内容了。所以前端页面开发完成后更新的内容将在用户再一次访问时才会生效，而不是马上就能生效。通常，一个基本的 Application Cache 离线页面应用至少应该包括 HTML 页面的 manifest 配置引用与被引用的 manifest 文件。

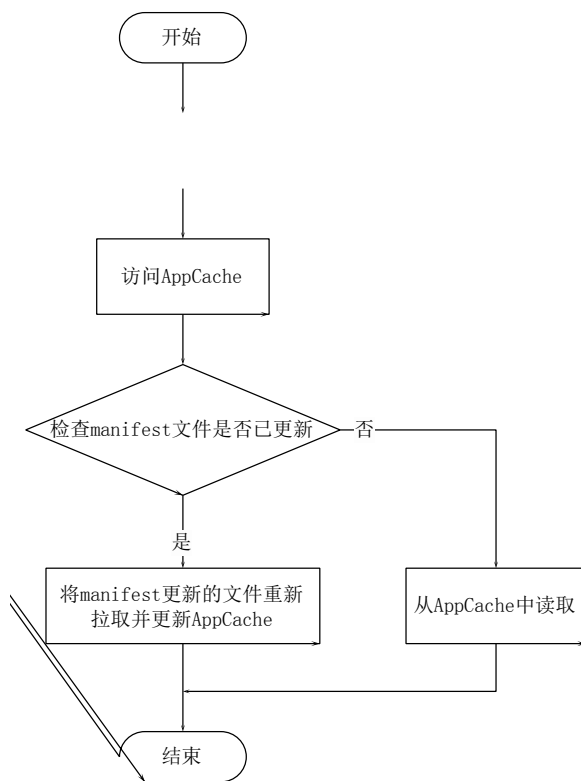


图 1-14 Application Cache 文件访问与更新机制

```

<!-- index.html -->
<html manifest="app.manifest">
  <head>
    <title>AppCache Test</title>
    <link rel="stylesheet" href="main-abs93lpd.css">
  
```

```
<script src="main-9id3dlsj.js"></script>
</head>
<body>
  <div id="main"></div>
</body>
</html>
```

对应的 manifest 描述文件如下：

```
CACHE MANIFEST
#VERSION 1.0
CACHE:
main-abs93lpd.css
main-9id3dlsj.js
```

假设这里 main-9id3dlsj.js 和 main-abs93lpd.css 为外部引用的文件，HTML 下载时会根据 manifest 内容去加载 Application Cache 中 CACHE 列表下的文件，如果服务器更新，CACHE 列表下的内容通常会发生变化并重新指向新的资源列表，浏览器根据此来进行 Application Cache 更新。另外需要注意的是，在更新缓存时，我们也可以通过 window.applicationCache 对象来访问浏览器的 Application cache，并可以查看对象的 status 属性来获取 cache 对象的当前状态。

```
let appCache = window.applicationCache;
switch (appCache.status) {
  case appCache.UNCACHED: // UNCACHED == 0, 表示未缓存
    return 'UNCACHED';
    break;
  case appCache.IDLE: // IDLE == 1, 表示闲置
    return 'IDLE';
    break;
  case appCache.CHECKING: // CHECKING == 2, 表示检查中
    return 'CHECKING';
    break;
  case appCache.DOWNLOADING: // DOWNLOADING == 3, 表示下载中
    return 'DOWNLOADING';
    break;
  case appCache.UPDATEREADY: // UPDATEREADY == 4, 表示已更新
    return 'UPDATEREADY';
    break;
  case appCache.OBSOLETE: // OBSOLETE == 5, 表示已失效
    return 'OBSOLETE';
    break;
  default:
    return 'UNKNOWN CACHE STATUS';
    break;
};
```

`applicationCache` 对象也可以主动更新 `Cache` 内容，不需要等到下一次更新。例如调用 `applicationCache.update()` 方法，这样浏览器可以去主动尝试更新用户的 `Cache`（在 `manifest` 文件已经改变的情况下）。最后，当 `applicationCache.status` 处于 `UPDATEREADY` 状态时，调用 `applicationCache.swapCache()` 方法，旧的 `Cache` 内容就会被置换成新的。

```
let appCache = window.applicationCache;
appCache.update(); // 尝试更新用户的 Application Cache
// TODO...
if (appCache.status == window.applicationCache.UPDATEREADY) {
  appCache.swapCache(); // 置换新的 Application Cache 内容
}
```

尽管 `Application Cache` 的实现很方便，但是仍然需要注意以下几个问题。

1. `Application Cache` 已经开始被标准弃用，渐渐将会由 `ServiceWorkers` 来代替，所以现在不建议使用 `Application Cache` 来实现离线应用，仅作为一种技术了解即可。

2. `Application Cache` 仍不能兼容目前全部主流的浏览器环境，即使是在移动端。

3. `Application Cache` 为站点离线存储提供的容量限制是 `5MB`，现在来说显然不适用。

4. 如果 `manifest` 文件或者内部列表中的某一个文件不能正常下载，整个更新过程将视为失败，浏览器将继续使用旧的缓存。

5. 引用 `manifest` 的 `HTML`、缓存列表的静态资源必须与 `manifest` 文件同源，即保持在同一个域下。

6. 站点中的其他页面即使没有设置 `manifest` 属性，请求的资源也会从缓存中访问。

7. 当 `manifest` 文件发生改变时，资源请求本身也会触发更新。

总之，`Application Cache` 仍是一个不成熟的本地缓存解决方案，实际项目中也并不推荐使用，但其设计思路为我们后面实现离线访问机制提供了方向。

## 👉 `cacheStorage`

`cacheStorage` 是在 `ServiceWorker` 规范中定义的，可用于保存每个 `ServiceWorker` 声明的 `Cache` 对象，是未来可能用来代替 `Application Cache` 的离线方案。`cacheStorage` 有 `open()`、`match()`、`has()`、`delete()`、`keys()` 五个核心 API 方法，可以对 `Cache` 对象的不同匹配内容进行不同的响应，`cacheStorage` 在浏览器端为 `window` 下的全局内置对象 `caches`，那么我们就可以直接通

过下面的形式来使用了。

```

caches.has();           // 检查如果包含 Cache 对象，则返回一个 promise 对象
caches.open();          // 打开一个 Cache 对象，并返回一个 promise 对象
caches.delete();        // 删除 Cache 对象，成功则返回一个 promise 对象，否则返回 false
caches.keys();          // 含有 keys 中字符串的任意一个，则返回一个 promise 对象
caches.match();         // 匹配 key 中含有该字符串的 cache 对象，返回一个 promise 对象

```

要了解 `cacheStorage`，我们必须深入了解一下 `ServiceWorker`，`ServiceWorker` 与 `WebWorker` 一样是在浏览器后台作为一个独立的线程运行的 `JavaScript` 脚本，可以为浏览器提供并行的计算和数据处理能力，并通过 `message/postMessage` 方法在页面之间进行通信，但是不能与前端界面进行交互。我们知道 `Native APP`（一般指移动客户端的原生应用）可以做到消息推送、离线使用、自动更新等，同样地，如果使用 `ServiceWorker` 也可以让 `Web` 应用具有类似功能。

图 1-15 为 `ServiceWorker` 运行的生命周期，我们在使用 `ServiceWorker` 时通常需要先注册 `ServiceWorker` 的脚本文件，然后在其脚本中运行 `caches` 的缓存控制方法，`caches` 是浏览器提供的用于存储文件缓存管理的对象，也是浏览器提供的 `cacheStrage` 全局对象。例如我们可以用如下方法在浏览器中注册一个 `ServiceWorker`。

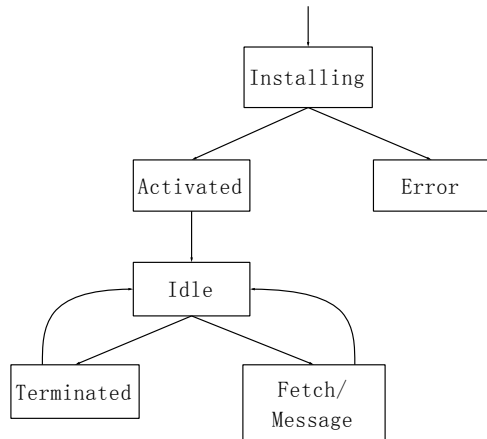


图 1-15 `ServiceWorker` 运行的生命周期

```

if (navigator.serviceWorker) {
  navigator.serviceWorker.register('./service-worker.js').then(function(registration)
  {
    console.log('service worker 注册成功');
  }).catch(function (err) {
    console.log('servcie worker 注册失败')
  });
};

```



```

}

// service-worker.js
let cacheList = [
  'main.js',
  'main.css'
];

// 这样就将缓存的文件列表注册到 CacheStorage 里面了，浏览器端使用 caches 全局变量 caches 来管理
this.addEventListener('install', function(event) {
  // 添加 cacheStorage 缓存
  event.waitUntil(
    caches.open('my-page-cache').then(function(cache) {
      return caches.addAll(cacheList);
    })
  );
});

```

这样就将 `cacheList` 中的两个文件路径注册到 `cacheStorage` 中了，通过监听 `ServiceWorker` 的 `fetch` 方法，如果匹配到这两个文件对应的 URL 路径请求，`cacheStorage` 就可以返回特定的本地缓存进行响应，而不用再向服务端发送请求。而且 `ServiceWorker` 脚本的更新也很简单，版本更新后注册引用新的文件就可以了。

```

// 根据 ServeWorker 返回 caches 中的文件，而不发送浏览器请求
this.addEventListener('fetch', function(event) {
  event.respondWith(caches.match(event.request).then(
    function(response) {
      // 如果有线上返回内容，则直接返回
      if (response) {
        return response;
      }
      let responseToCache = response.clone();
      // 否则读取缓存里面的响应返回
      caches.open('my-page-cache').then(function(cache) {
        cache.put(event.request, responseToCache);
      });
    })
  ));
});

```

通过这种方式实现离线缓存的优点是可以利用浏览器的本身机制来实现，而不需要过多服务端复杂的设计。然而我们也需要知道，目前 `ServiceWorker` 的浏览器兼容性很差，导致这种方案还不成熟，至少在短期内仍不是一个可行的方案。

## 👉 Flash缓存

Flash 缓存的方式目前应用比较少，它主要基于网页端 Flash，具有读写浏览器端本地目录

的功能，同时也可以向 JavaScript 提供调用的 API，这样页面就可以通过 JavaScript 调用 Flash 去读写指定的磁盘目录，达到本地数据缓存的目的。

关于浏览器缓存实现的技术和方式较多，尤其是在较新的浏览器上，但我们仍要明白目前可以在项目中配置应用的只有 HTTP 缓存、localStorage 和 Cookie。ServiceWorker 在将来可能会被使用，但目前兼容性的欠缺仍然不能忽视，其他的缓存方式我们仅作为知识了解即可。

参考资料：

<http://www.html5rocks.com/en/tutorials/webdatabase/websql-indexeddb/>。

<http://www.html5rocks.com/en/tutorials/offline/storage/>。

<http://www.html5rocks.com/en/tutorials/service-worker/introduction/>。

### 1.3 前端高效开发技术

前端项目工程随着网站项目或 Web App 项目的出现而产生，随着项目越来越庞大，前端工程的开发工作也越来越复杂。对于前端工程师来说，掌握和运用什么样的前端开发技术直接决定着前端项目实践中的工程开发效率是什么样的。那么前端开发技术具体指的是什么呢？通俗地说，前端开发技术指的是前端开发实践过程中所用到的一切工程方法和手段。

这一节，我们就来简单聊一聊与前端开发技术相关的内容。

#### 1.3.1 前端高效开发工具

在前端实践中，任何项目的开发仍然需要从最基本的编辑器说起。工欲善其事，必先利其器。高效便捷的开发工具能帮助我们提高编码的效率，避免不必要的时间消耗。就前端而言，主流的开发工具主要有 Sublime、Webstorm、Vscode、Vim 等，你可能要说这里没有你认为优秀的编辑工具，我们先以这四种前端开发工具为例做一个简单的比较，后续继续讨论其他工具。

表 1-3 中简单列举了一些主流的前端开发编辑工具及其优缺点。

表 1-3 前端常用开发工具

编 辑 器	优 势	不 足
sublime	较轻量级、插件齐全的开发工具，扩展插件工具覆盖非常全面	没有自带 debug 和断点功能
webstorm	集成较全面的开发工具，也可以选择扩展，可使用命令行或断点	相对较重量级
vscode	较轻量级，原生支持 TypeScript，关联断点调试非常方便，也可扩展	相对完整性稍弱

续表

编辑器	优势	不足
vim	Linux 下可选的高效工具	入门相对难，没有较多前端对应的高效辅助插件

无论选择哪一种，一款高效的开发工具都应该具备以下几个方面的能力。

- 代码格式化 **Format** 能力。编辑器具有自动按照默认的设置或人为设定的规范格式化 **HTML/JavaScript/CSS** 等语法代码的能力，避免我们花费时间去调整不规范的格式。
- 代码模板 **Snippet** 能力。例如，快捷键 + **tab** 具有自动生成 **for** 循环或 **try...catch** 代码块的能力，避免我们手动输入重复代码或注释而花费大量的时间。
- 自动检测错误能力。例如，各种代码 **Lint**、**Hint** 工具自动提示 **HTML**、**JavaScript**、**CSS** 不规范的地方，辅助我们快速开发，不用自己查找一些低级的语法错误或不规范的地方。
- 编辑快捷键能力。这应该是任何一个高效的工具都要具备的能力，而且快捷键应该越全越好。
- 自动 **Debug** 能力。现在大多浏览器能提供这一功能，每个人也可以按需选择，最新的 **Chrome** 也支持 **ECMAScript 6** 以上的调试，如果编辑器能具备这个能力当然再好不过。
- **Git** 或 **Svn** 版本控制插件能力。这个功能开发者可以根据需要自行选择，关于 **Git** 或 **Svn** 的使用这里不做阐述。
- 自动文档工具。开发的过程中应尽量保留文档，但是我们很可能没有太多时间写文档，因此，如果编辑器插件能帮助我们自动高效生成代码文档注释就比较好了。另外要注意的是，自文档化代码的编写也是一个很好的习惯。

很多时候编辑器其实无法满足上述所有的功能，那就要根据我们自己的核心需求进行选择了。每个人的核心需求可能不同，选择也不同，但无论怎样，好的工具的核心特点是能够帮助我们实现高效开发，而不是工具本身具备的功能，过多地关注工具本身就舍本逐末了。

在我们开始敲键盘之前，最好确认编辑工具是否具有以上这几点高效的特性。但如果你还没有使用到这些扩展的辅助功能，那么请赶紧去完善工具，好的开发工具不仅可以提高我们开发的速度，也可以辅助我们写出更高质量的代码。

## 1.3.2 前端高效调试工具

### 前端快速调试工具Chrome浏览器

通常前端开发过程中使用最多的调试工具大概就是 Chrome 浏览器了，早期使用 Firefox 浏览器较多，不过现在使用的人很少，只是偶尔会考虑页面在 Firefox 浏览器上的兼容性问题。

虽然 Chrome 只是一款浏览器，但是要了解使用 Chrome 所有的开发调试技巧也是很难的。另外还有一些高效的开发者工具插件，如 Postman、hostAdmin 等。图 1-16 为 Chrome 浏览器的调试工具界面。

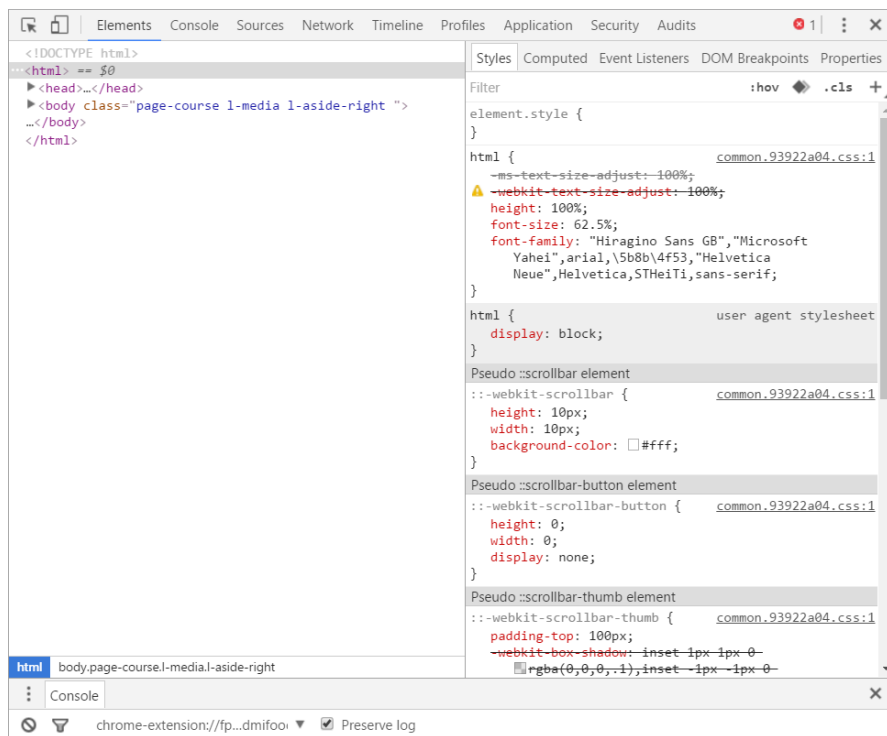


图 1-16 Chrome 浏览器调试工具界面

在 Chrome 浏览器中，用 F12 打开控制台后，一名优秀的前端工程师需要尽量保证自己对里面 95% 以上的操作和内容都很熟悉，这样才可以说是基本运用自如。Chrome 的调试面板主要包括设备模拟、Elements、Console、Sources、Network、Timeline、Profiles、Resources、Security、Audits 这些内容。

- 设备模拟不仅可以让 Chrome 模拟移动端浏览器和桌面浏览器中打开页面的情况，而

且还可以模拟移动端中常见的不同机型屏幕大小和分辨率情况下加载页面的显示结果，或者自己添加特定模拟设备的屏幕来模拟显示效果。

- Elements 则可以用于阅读 DOM 结构和 DOM 样式的内容及规则，或者添加阻塞 DOM 渲染的断点功能来看页面的渲染细节。
- Console 的功能是查看控制台输出的内容或者直接执行某部分 JavaScript 脚本的运行命令行。
- 通过 Sources 我们可以浏览网站加载的所有静态目录资源，同时可以进行资源内容的查看阅读和脚本的断点调试。
- NetWork 常用于查看页面内容加载的网络请求情况，如请求的返回码、类型、大小、消耗时间、网页资源加载时序图等。
- 使用 Timeline 可以查看浏览器执行性能和内存消耗的时序图情况。
- Profiles 是测试网页性能消耗的一种方式，用于统计例如 CPU 等系统资源在页面加载过程中的消耗分布情况。
- Application (旧版的 Chrome 浏览器中为 resources) 用于查看 Chrome 浏览器缓存情况，主要包括 HTTP 文件缓存、Application Cache、ServiceWorker 缓存对象、LocalStorage、SessionStorage、IndexedDB、WebSQL、Cookies、Cache Storage 等缓存空间中的情况。
- Security 用于管理网站安全证书，例如，在 HTTPS 的网站下面我们可以通过它来查看网站使用的证书内容。
- Audits 则可以根据目前页面文档加载和脚本执行情况给出当前前端页面的部分优化建议，这对于前端页面的优化具有极其重要的借鉴意义。

除了普通的 Debug 和模拟移动设备这些功能，Chrome 还提供了移动连接模拟真实设备的 inspect 查看功能。例如，在 Chrome 地址栏中输入 `chrome://inspect/#devices` 即可查看主机当前连接的移动设备浏览器打开网页的情况，并可以阅读 DOM 结构和查看 Debug 信息。Chrome 还有很多有意思的扩展功能，打开 `chrome://chrome-urls/` 就可以知道所有 Chrome 支持的扩展功能信息，包括浏览器的各类工具界面入口。

```
chrome://version/    //查看系统信息
chrome://inspect/    //查看连接设备调试信息
chrome://downloads  //浏览器下载管理
```

```
chrome://settings/ //浏览器设置
...
```

当然，只会使用 Chrome 浏览器调试是远远不够的，使用 Chrome 可以帮助我们便捷快速地定位大部分脚本逻辑或页面性能问题，但是如果开发多浏览器下的兼容页面，还必须要考虑兼容 Internet Explorer、Firefox 或移动端各种版本真实环境的浏览器，所以这时 Chrome 通常只作为项目前期高效开发的调试工具，开发完成后仍需要考虑更多不同浏览器版本的兼容性问题。

## 📌 网络辅助工具

除了前端调试工具，我们通常还需要依靠一些辅助工具来协助开发。前端开发中必须要提到的一个辅助开发工具就是 Fiddler，没有使用过的建议尝试下。其基本原理是作为本地的一个代理服务，将特定的应用层网络请求拦截，来模拟需要的不同场景。拦截处理规则可以由使用者来制定。其实很像一个本地的 Nginx 服务器，对配置的域名或地址请求返回对应的模拟响应内容。除了获取本机的网络请求，Fiddler 还可以作为代理拦截其他连入设备的请求，这样我们在移动端进行开发时，真实设备上的请求就可以通过配置 Fiddler 代理来获取了。

如图 1-17 所示，将移动设备的网络配置代理指向一台运行 Fiddler 的开发计算机，那么移动设备发起的 HTTP 请求都会经过开发计算机被 Fiddler 代理拦截，然后便可以直接配置执行该请求的返回。例如将 `http://www.domain.com/text.html` 的 URL 请求返回开发计算机中磁盘的 `text.html` 文件内容，或者模拟访问 `http://www.domain.com/index.htm` 地址返回 404 的情况。除此之外，Fiddler 也可以查看请求资源内部信息，查看资源加载时序图，模拟在慢速网络下查看页面内容输出和调试的情况等。有人基于 Fiddler 做了一个 Willow 的插件来实现更强的代理替换规则，例如，路径目录替换、特定 DNS 域名替换、HOST 配置等。总的来说，Fiddler 目前已经成为前端开发中不可或缺的辅助开发工具之一。



图 1-17 Fiddler 设置代理工作流程

## 📌 Node调试工具

讲 Node 调试工具是因为 Node 端开发在本书中被列为前端知识内容的一部分，后面的章节

会展开介绍，所以这里我们先简单了解一下 Node 的开发调试方式。

服务端开发调试的工具也比较多。例如 node-supervisor、node-inspector 及以后可能出现的新工具。这类工具入门很简单，按照参考文档安装后，用它们特定的命令启动应用入口文件就可以了。举例来说，node-inspector 安装如下。

```
$ npm install -g node-inspector
$ node-debug app.js
Node Inspector is now available from http://127.0.0.1:8080/?ws=127.0.0.1:8080&port=5858
Debugger listening on port 5858
```

这样要调试的 Node 端服务就通过 debug 模式启动了，我们还需要用 node-inspector 服务将运行的代码同步到浏览器端进行调试，因此还需打开另一个终端。

```
$ node-inspector
Node Inspector v0.11.2
Visit http://127.0.0.1:8080/?ws=127.0.0.1:8080&port=5858 to start debugging.
```

这时 node-inspector 会启动一个远程可访问的网络端口服务，开发者通过浏览器可以对 Node.js 的文件进行查看同时断点调试。访问 <http://127.0.0.1:8080/?ws=127.0.0.1:8080&port=5858> 会进入如图 1-18 所示的一个文件目录的调试界面，然后我们就可以进行断点调试了。

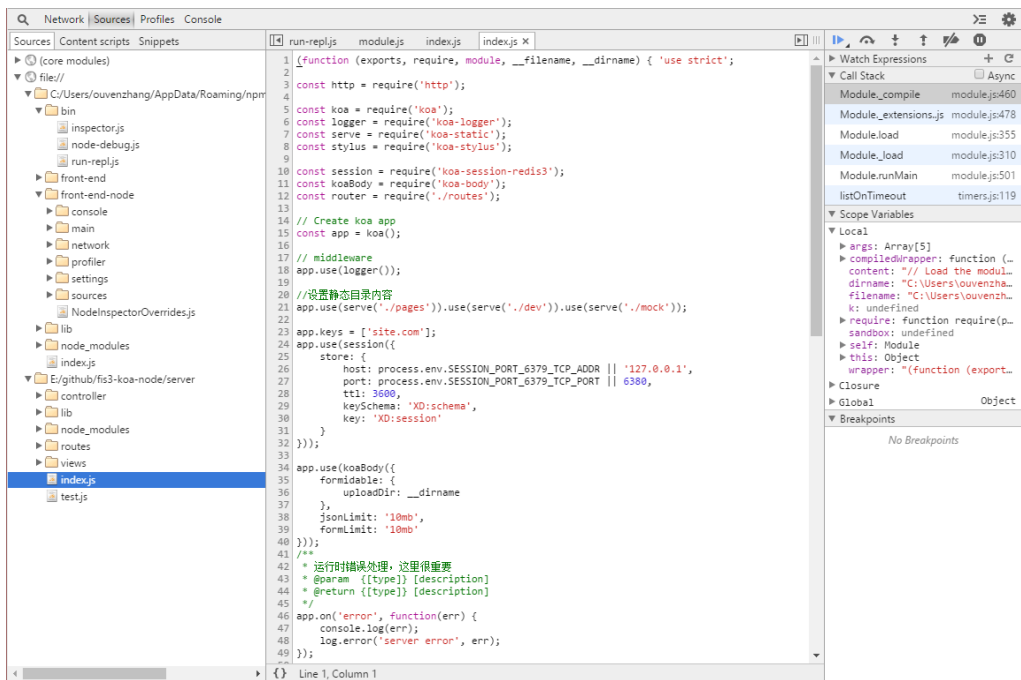


图 1-18 node-inspector 浏览器端调试界面

如图 1-19 所示，之所以能这样做是因为 node-inspector 这类调试工具可以将 Node 服务的 JavaScript 代码解析后通过自身开启的端口发送到浏览器端运行，这样可以将 Node 端的 JavaScript 的逻辑映射到浏览器中控制执行，实现 Node 端执行代码和浏览器载入代码同步调试功能。当然，调试时一般推荐使用较新版本的 Chrome 浏览器。

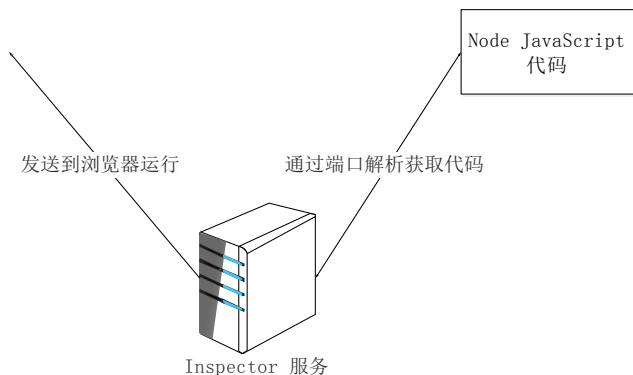


图 1-19 Inspector 远程调试运行原理

除此之外，Node 端开发也可以同其他服务端语言开发一样使用更直接的方式，例如通过输出 log 日志文件或信息来进行 Node 端的开发调试等方式。

### 前端远程调试工具

在前端页面开发中，除了一般的开发调试方法，也有一些例如 Vorlon.js、Weinre 等用于移动端浏览器的远程调试工具。前端远程调试工具的原理和使用方式跟 node-inspect 类似，也需要启动一个调试代理服务将远程设备上的代码发送到开发机器的模拟浏览器上逐行执行，同时开发机模拟浏览器上的操作也要回馈给远程设备。例如，使用 vorlon 来进行远程调试就可以通过以下代码来实现。

```
$ npm install -g vorlon
$ vorlon
2016-12-12 13:6:57 - info: Vorlon.js PROXY listening on port 5050
2016-12-12 13:6:57 - info: Vorlon.js SERVER listening on port 1337
```

此外，我们还需要在页面中引用一个 JavaScript 文件来支持浏览器端 vorlon 的调试服务。

```
<script src="http://localhost:1337/vorlon.js"></script>
```

这样页面上就可以显示如图 1-20 所示的浏览器端远程调试的界面了。



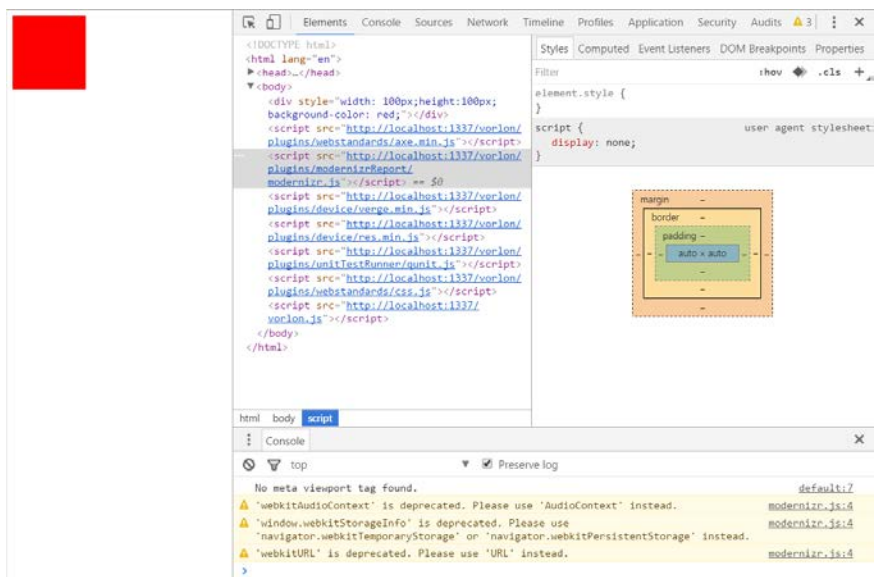


图 1-20 Vorlon 远程调试运行界面

例如在开发机浏览器上选中一个`<div>`元素，这时 Vorlon 或 Weinre 的 Node 端服务会通知将远程设备上对应的这个元素加上一个高亮边框来表示该元素被选中。相对于 node-inspect，这里的远程设备可能是真实的移动端设备，而不是 Node.js 环境。

如图 1-21 所示，调试代理服务器可以将远程设备浏览器中的内容加载到开发机上重新解析，开发机浏览器上的操作也会通过调试代理服务的特定端口同步到远程设备上响应，这样就可以在开发机浏览器上实时查看远程设备浏览器上显示的内容了，而且开发机模拟浏览器上的操作也可以映射同步到远程设备上。

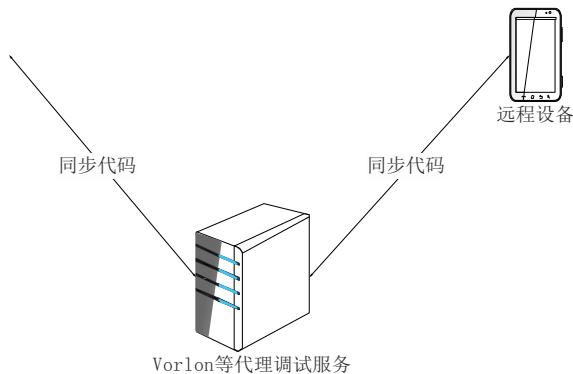


图 1-21 前端远程调试运行原理

基于项目实践经验，这种调试方式虽然看起来比较高效，但使用时问题较多，而且代理调试服务器由于频繁进行同步可能会导致内容不稳定也是一个问题。例如，有时同步刷新的实时性不强，或者直接没反应。所以大家可以了解这种调试方式，必要的时候作为备用方案使用，从实际开发的角度上来讲，还是尽量先使用 Chrome 配合 Fiddler 辅助工具的方式来高效快速地定位大部分问题。

除了上面介绍的开发调试方法，还有一些其他的可选调试方案。实际开发中我们可能并不会都用到，具体要根据实际场景和个人习惯来选择。更多时候我们遇到的问题是可控的，大多数问题也不是必须使用某个特定的调试方法才能重现解决的，所以在开发过程中还要灵活选择运用。

## 1.4 本章小结

这一章主要介绍了前端技术的发展概况以及一些必须掌握的浏览器基础知识与常用开发技术。总体来说，对前端技术的发展和前景有一个正确的认识是作为前端工程师最基本的素养，另外，掌握浏览器缓存技术也能帮助我们整体把握持久化存储的实现概貌，有利于我们根据实际的应用场景选择合适的缓存方案。作为前端工程师，我们需要做的事情还有很多，这也是现代前端赋予我们的机遇和挑战。下一章中，我们将对前端相关的协议进行分析总结，让读者从基础概念和实践中理解与前端相关的各类协议的原理与实现。

# 第 2 章

## 前端与协议

我们知道，浏览器上解析执行的 HTML、CSS 和 JavaScript 文件通常是通过网络请求从 Web 服务器上下载解析的，加载过程中，浏览器通过网络模块创建下载进程，发起 HTTP 请求，将 HTML 文本、CSS 样式或 JavaScript 脚本装载进浏览器解析或运行。这里就涉及了一些相关的网络协议，我们可以认为与前端关系最密切的协议是 HTTP 协议，因为几乎所有的前端相关资源文件均是通过 HTTP 协议请求完成的。前端 Web 应用的发展加速了前后端技术的分离，这种开发模式降低了前端与服务端的耦合，但是前端和服务端之间的交互数据通信仍是通过协议来完成的，这里的协议可以认为是前后端开发者之间主观协商形成的一层数据接口规范。当然，还有基于 SSL（Secure Sockets Layer，安全套接字层）层的 HTTPS 协议。进入移动互联网时代后，移动端 Web 脚本开始需要与移动端原生程序进行交互，这便涉及与移动端 Native 原生程序交互的协议。除了这些还有 HTML5 的 WebSocket 实时通信协议、与服务端交互的 RESTful 协议等。

可见，各类协议在前端开发中被应用于方方面面，那么在这一章中，我们就一起来看看与前端开发密切相关的协议。

### 2.1 HTTP协议简介

#### 2.1.1 HTTP协议概述

HTTP（HyperText Transport Protocol，超文本传输协议）协议是 WWW 服务器和用户请求代理（例如浏览器等）之间通过应答请求模式传输超文本（例如 HTML 文件、JavaScript 文件、CSS 文件、图片甚至服务器接口数据等）内容的一种协议，协议的详细规范序号为 RFC2616。

图 2-1 为某用户使用浏览器请求 `http://www.jixianqianduan.com/` 首页和应答时 HTTP 消息内容的传递过程。浏览器（用户请求代理）向服务器发送请求时头部中包含请求的方法 GET、URL（Uniform Resource Location，统一资源定位符）`http://www.jixianqianduan.com/`、协议版本号 1.1、请求头域字段（如请求接受类型 Accept）、缓存控制 Cache-Control、浏览器 Cookie 和 user-Agent 信息等，同时也可能会带上请求的正文内容。服务器接收请求处理后也是以一段响应报文作为返回，响应的内容包括 HTTP 消息响应的协议版本 1.1、返回码 304 及返回描述 Not Modified、缓存控制信息 Cache-Control 以及正文的 HTML 内容等，当然如果返回码为 304 时请求响应返回的正文为空，浏览器将从本地缓存中读取文件。浏览器接受到服务器的返回后会进行解析，同时进行相应的操作，例如将服务器返回正文中的 HTML 内容装载至浏览器进行解析渲染，或是将服务器返回的 JSON 字符串解析成前端可用的 JSON 对象等。



图 2-1 HTTP 应答过程

通常一个完整的 HTTP 报文由头部、空行、正文三部分组成。空行用于区分报文头部和报文正文，由一个回车符和一个换行符组成。图 2-2 是一个 HTTP 请求报文的格式结构：请求头部通常由请求类型、请求 URI、协议版本和扩展内容组成；请求头中还包含其他请求头域信息，如 Accept、Cookie、Cache-Control、Host 等；请求正文可以携带浏览器端请求的内容，如

POST、PUT 请求的表单内容。响应返回报文的格式与此类似，图 2-3 为服务器的响应报文结构：响应报文头部由状态码、状态描述、协议版本、扩展内容组成；响应头包含响应头部域信息，如 Date、Content-Type、Cachel-Control、Expires 等；服务器返回给浏览器的信息可以放在报文正文部分。

请求URI	协议版本	扩展内容
请求头部域内容：Accept\Cookie\Cache-Control\Host 等		
空行		
正文内容		

图 2-2 HTTP 请求头部结构

状态描述	协议版本	扩展内容
响应头部域内容：Date\Content-Type\Cache-Control\Expires 等		
空行		
正文内容		

图 2-3 HTTP 响应头部结构

HTTP 协议自出现到现在，先后经历了 HTTP 0.9 版本、HTTP 1.0 版本、HTTP 1.1 版本和 HTTP 2 版本这四个版本。不过，目前使用最为广泛的仍然是 HTTP 1.1 版本。HTTP 1.1 标准发布于 1999 年，相对于 HTTP 1.0 增加了协议扩展切换、缓存、部分文件传输优化、长连接、消息传递、host 头域、错误提示等一些重要的增强特性。接下来我们就简单了解一下 HTTP 1.1 版本中的几个重要特性和相关的应用场景。

### 2.1.2 HTTP 1.1

#### 👉 长连接

提到 HTTP 1.1 协议，我们首先想到的一个重要特性就是长连接。HTTP 1.1 的长连接机制是通过请求头中 keep-alive 头域信息来控制的。HTTP 1.0 默认请求的服务器返回是没有

keep-alive 的，但在 HTTP 1.0 中，如果要建立长连接，也可以在请求消息中包含 `Connection: keep-alive` 头域信息，如果服务器能识别这条连接的头域信息，则会在响应消息头域中也包含一个 `Connection: keep-alive` 返回，表示后面的文件请求可以复用之前的连接传输。但是在 HTTP 1.1 协议中，任何 HTTP 请求的报文头部域都会默认包含 keep-alive。keep-alive 的控制可以让客户端到服务器端之间的连接在一段时间内持续有效，当一个请求文件的传输连接建立以后，在服务器保持该连接的这段时间内，其他文件请求可以复用这个已经建立好的连接，而不用像 HTTP 1.0 那样重新握手建立连接，这样就有效将建立和关闭连接的网络开销平均到多个文件的请求上。例如有  $n$  个文件的连续请求，每个文件请求建立和关闭连接的开销为  $ams$ （毫秒），那么完全使用 HTTP 1.0 协议完成所有文件请求的额外开销就为  $n \times ams$ ，而使用 HTTP 1.1 协议来进行文件传输额外开销仅为  $ams$ 。但是需要注意的是，长连接请求机制并不会节省传输内容的网络开销。

### 📌 协议扩展切换

协议扩展切换是指，HTTP 1.1 协议支持在请求头部域消息中包含 `Upgrade` 头并让客户端通过头部标识令服务器知道它能够支持其他备用通信协议的一种机制，服务器根据客户端请求的其他协议进行切换，切换后使用备用协议与客户端进行通信。例如 WebSocket 协议就是典型的应用，WebSocket 协议通信是通过 HTTP 的方式来建立的，通信连接建立完成后通知服务器切换到 WebSocket 协议来完成后面的数据通信。

图 2-4 为 WebSocket 协议的连接建立过程，浏览器（假设用户使用的浏览器支持 WebSocket）向服务端发送请求，并在消息头中添加 `Connection: Upgrade` 和 `Upgrade: websocket` 告诉服务器后面需要进行协议切换，切换成为 WebSocket 协议进行通信，如果服务端支持 WebSocket 服务并允许该客户端来连接，则可以在响应报文头中返回 `Upgrade` 和 `Connection` 消息头域，同意浏览器使用 WebSocket 来连接，同时返回的状态码为 101 表示请求还需要完成协议的切换。



图 2-4 WebSocket 通信建立过程

### 缓存控制

在 HTTP 1.1 版本之前，浏览器缓存主要是通过对 HTTP 1.0 的 Expires 头部控制来实现的，我们知道 Expires 只能根据绝对时间来刷新缓存内容，HTTP 1.1 增加了 Cache-Control 头域，可以支持 max-age 用来表示相对过期时间，另外请求服务器时也可以根据 Etag 和 Last-Modified 来判断是否从浏览器端缓存中加载文件，此时缓存的控制和判断将决定服务器的响应报文中头部内容的状态码 200 还是 304。下面来看一个浏览器发送 HTTP 请求时进行缓存读取判断的流程。

如图 2-5 所示，浏览器发起请求时，头部域字段的判断过程主要如下所述。

1. 浏览器会先查询 Cache-Control（这里用 Expires 判断也是可以的，但是 Expires 一般设置的是绝对过期时间，在 HTTP 1.1 之前较为通用，Cache-Control 设置的是相对过期时间，HTTP 1.1 后推荐使用 Cache-Control 来控制，如果两者都设置了，则只有 Cache-Control 的设置生效）来判断内容是否过期，如果未过期，则直接读取浏览器端缓存文件，不发送 HTTP 请求，否则进入下一步。

2. 在浏览器端判断上次文件返回头中是否含有 Etag 信息，有则带上 If-None-Match 字段信

息发送请求给服务器，服务端判断 Etag 未修改则返回 304，如果修改则返回 200，否则进入下一步。

3. 在浏览器端判断上次文件返回头中是否含有 Last-Modified 信息，有则带上 If-Modified-Since 字段信息发送请求，服务端判断 Last-Modified 失效则返回 200，有效则返回 304。

4. 如果 Etag 和 Last-Modified 都不存在，则直接向服务器请求内容。

这就是 Cache-Control、Etag 和 Last-Modified 控制请求缓存的主要过程。

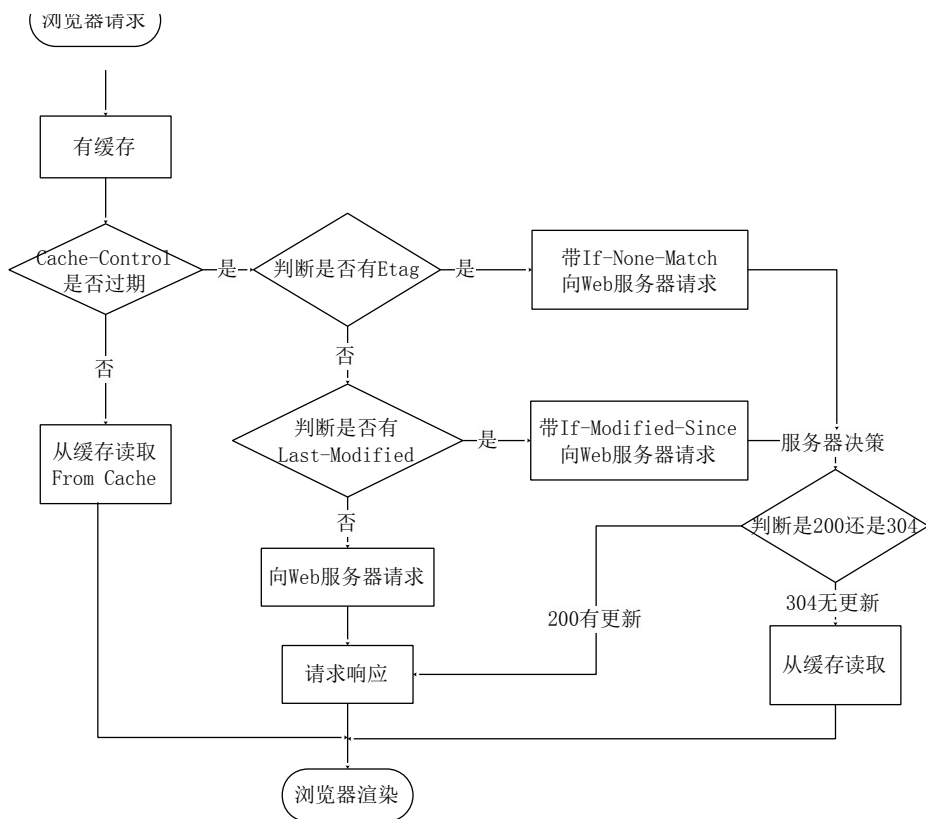


图 2-5 浏览器请求缓存判断过程

## 部分内容传输优化

部分内容传输优化指 HTTP 可以支持超文本文件的部分传输，例如，它允许请求一个文件的起始位置和一个偏移长度来进行文件内容的部分传输。



另外 HTTP 1.1 请求允许携带一些数据参数信息一起发送到服务器, 请求时的数据信息可以放在请求头 (例如, GET、DELETE 方法请求时) 或正文 (例如, POST、PUT 方法请求时) 中。HTTP 请求在消息的正文中除了可以携带文本内容, 也可以传输二进制数据, 例如表单中使用 formData 提交上传文件时携带的就是二进制数据。

HTTP 报文的头部域信息内容其实有很多, 每个头部域字段的控制都具有自己的逻辑和判断机制, 以下是常见的一些头部域字段的设置。

1. Accept: 告诉 Web 服务器自己能接收什么媒体类型, /\*/\*表示能接收任何类型, type/\*表示接收该类型下的所有子类型, 一般格式为 type/sub-type, 多个类型使用 q 参数分割, q 的值代表 quality 请求质量, 反映了用户对这类媒体类型的偏好程度, 例如 Accept: text/plain; q=0.5, text/html, text/x-dvi; q=0.8, text/x-c。

2. Accept-Charset: 浏览器接收内容的字符集, 通常是 utf-8。

3. Accept-Encoding: 浏览器接收内容的编码方法, 例如指定是否支持压缩, 若支持压缩的话支持什么压缩方法, 具体如 Accept-Encoding:gzip, deflate, sdch。

4. Accept-Language: 浏览器接收内容的语言。语言跟字符集是有区别的, 例如中文是语言, 中文有多种字符集, big5、gb2312、gbk 等。该参数也可以设置多个, 如 Accept-Language: zh-CN,zh;q=0.8。

5. Accept-Ranges: Web 服务器表明自己是否接受获取某个实体的一部分 (比如文件的一部分) 请求, 这里主要用于部分文件传输, 实际上我们用的比较少。bytes 表示接受传输多大长度内容, none 表示不接受。

6. Age: 一般当服务器用自己缓存的实体去响应请求时, 可以用该头部表明实体从产生到现在经过了多长时间, 如 Age: 3600。

7. Allow: 该参数头部可以设置服务端支持接收哪些可用的 HTTP 请求方法, 例如 GET、POST、PUT, 如果不支持, 则会返回 405(Method Not Allowed)。

8. Authorization: 当客户端接收到来自 Web 服务器的 WWW-Authenticate 响应时, 后面可以用该头部来携带自己的身份验证信息给 Web 服务器直接进行认证。

9. Cache-Control: 用来声明服务器端缓存控制的指令。包括请求设置指令和响应请求指令。

请求控制指令如下。

no-cache: 不使用缓存实体, 要求从 Web 服务器去请求内容。

max-age: 只接受 Age 值小于 max-age 值的内容, 即没有过期的请求对象。

max-stale: 可以接受过去的对象, 但是过期时间必须小于 max-stale 值。

min-fresh: 接受生命期大于其当前 Age 跟 min-fresh 值之和的缓存对象。

响应控制指令如下。

public: 可以用 Cache 中内容回应任何用户。

private: 只能用缓存内容回应先前请求该内容的具体用户。

no-cache: 可以设置哪些内容不被缓存。

max-age: 设置响应中包含对象的过期时间。

ALL: no-store 不允许缓存。

10. Connection: 在请求头中, close 告诉 Web 服务器或者代理服务器, 在完成本次请求响应后断开连接, 无须等待本次连接的后续请求, keep-alive 告诉 Web 服务器或者代理服务器, 在完成本次请求响应后保持连接, 等待本次连接的后续请求。在响应头中, close 连接已关闭。keep-alive 保持连接, 等待本次连接的后续请求, 如果浏览器请求保持连接, 则该头部表明希望 Web 服务器保持连接的时长(秒), 例如, keep-alive: 300。

11. Content-Encoding: 与请求头的 Accept-Encoding 对应, 指 Web 服务器表明使用何种压缩方法(gzip, deflate)压缩响应中的对象, 例如, Content-Encoding: gzip。

12. Content-Language: 与请求头中的 Accept-Language 对应, Web 服务器告诉浏览器响应的媒体对象语言。

13. Content-Length: Web 服务器告诉浏览器 HTTP 请求内容的长度。例如, Content-Length: 1024。

14. Content-Range: Web 服务器表明该响应包含的部分对象为整个对象的哪个部分。

15. Content-Type: 与请求头的 Accept 对应, 指明 Web 服务器告诉浏览器响应的对象的类型。例如, Content-Type: application/xml。

16. Etag: 对象(比如 URL)的标志值。一个对象(如 HTML 文件)如果被修改了, 其 Etag 也会被修改, 所以 Etag 的作用和 Last-Modified 差不多, 主要供 Web 服务器判断一个对象是否改变。例如前一次请求某个 HTML 文件时获得了其 Etag, 当这次又请求该文件时, 浏览器就会把先前获得的 Etag 值发送给 Web 服务器, 然后 Web 服务器会将这个 Etag 值跟该文件当前的 Etag 值进行对比, 判断文件是否改变。

17. Expires: Web 服务器表明该实体将在什么时候过期, 对于过期的对象, 只有在跟 Web 服务器验证了其有效性后, 才能用来响应客户请求, 是 HTTP/1.0 的头部。例如, Expires: Sat, 23 May 2009 10:02:12 GMT。

18. Host: 客户端指定自己访问的 Web 服务器的域名/IP 地址和端口号。例如, Host: www.jixianqianduan.com。

19. If-None-Match: 如果上次文件返回头中包含 Etag 信息, 则会带上 If-None-Match

发送请求给服务器，判断请求返回 304 还是 200。例如，If-None-Match: W / "34b1c4d4f5d8c61:d23"。

20. If-Modified-Since: 如果上次文件返回头中包含了 Last-Modified 信息，则会带上 If-Modified-Since 发送请求给服务器，判断请求返回 304 还是 200。例如，If-Modified-Since: Thu, 10 Apr 216 09:14:42 GMT。

21. If-Range: 浏览器告诉 Web 服务器，如果请求的对象没有改变，就把修改的部分返回给浏览器，如果对象改变了，就把整个对象返回给浏览器。浏览器通过发送请求对象的 Etag 或者自己所知道最后修改时间给 Web 服务器，让其判断对象是否改变。If-Range 常常跟 Range 头部一起使用。

22. Last-Modified: Web 服务器设置的对象最后修改时间，比如文件的最后修改时间或者动态页面的最后产生时间。例如 Last-Modified: Tue, 06 Sep 2016 02:42:43 GMT。

23. Location: Web 服务器告诉浏览器，试图访问的对象已经被移到别的位置了，让浏览器重定向去读取 Location 里面返回的内容。

24. Pragma: 主要使用 Pragma: no-cache, 相当于 Cache-Control: no-cache。例如，Pragma: no-cache。

25. Proxy-Authenticate: 代理服务器响应浏览器，要求其提供代理身份验证信息。Proxy-Authorization: 浏览器响应代理服务器的身份验证请求，提供自己的身份信息。

26. Range: 浏览器告诉 Web 服务器自己想读取对象的哪部分。

27. Referer: 浏览器向 Web 服务器表明自己是从哪个网页 URL 跳转访问当前请求中网址 URL 的，即跳转到当前页面的来源。例如，Referer: http://www.jixianqianduan.com/。

28. Server: Web 服务器通过此头域表明自己是什么软件及版本信息。例如，Server: Apache/2.2.18 (Unix)。

29. User-Agent: 浏览器的代理名称，位于请求头部，通常服务端可以根据这个设置获取浏览器的种类和版本信息。例如，Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36。

30. Transfer-Encoding: Web 服务器表明自己对本响应传输消息体做怎样的编码，如是否分块 (chunked)，Transfer-Encoding: chunked。

参考资料: [https://datatracker.ietf.org/doc/rfc2616/?include\\_text=1](https://datatracker.ietf.org/doc/rfc2616/?include_text=1)。

### 2.1.3 HTTP 2

HTTP 2 即超文本传输协议 2.0 版本，是 HTTP 协议的下一个版本。为了更好地了解 HTTP 2，

我们先来了解一下 SPDY 协议。SPDY 是一种基于 HTTP 的兼容协议，由 Google 发起，Chrome、Opera、Firefox 等较新的浏览器已提供该协议支持。SPDY 传输支持多路复用和服务器推送技术，压缩了 HTTP 头部减小了请求大小，并强制使用 SSL 传输协议，到目前为止已经成为了一套成熟的高效协议标准。但由于 SPDY 必须使用 HTTPS 协议，所以之前 HTTP 的网站就无法直接使用 SPDY，因此最终 HTTP Working-Group 决定以 SPDY 2 版本协议规范为基础，开发 HTTP 2 协议，能将文件内容在网络中进行高效传输，同时希望使用 HPACK 算法（为 HTTP 2 头压缩专门设计的算法）来压缩协议的消息头。最终形成了目前广为人知并且极具优势的下一代超文本传输协议 HTTP 2。

HTTP 2 相对于之前的 HTTP 协议有以下几个优点。

- HTTP 2 采用完全二进制的格式来传输数据，而非 HTTP 1.x 的默认文本格式。而二进制在网络中传输的基本单位一般为帧（Frame，一个帧可以理解为具有固定格式和长度的二进制数据包），每个帧包含几个固定部分内容：类型 Type、长度 Length、标记 Flags、流标识 Stream 和 Frame payload（帧有效载荷，一帧能携带的内容数据长度）。多个帧的传输在网络中就形成了帧的传输网络流，所以我们可以理解为 HTTP 2 协议是通过流式传输的。同时 HTTP 2 对消息头采用 HPACK 压缩传输，最大限度地节省了传输带宽。相比于 HTTP 1.x 每次请求都会携带大量冗余头信息（例如浏览器 Cookie 信息等），HTTP 2 就具有很大的优势了。
- HTTP 2 使用 TCP 多路复用的方式来降低网络请求连接建立和关闭的开销，多个请求可以通过一个 TCP 连接来并发完成。HTTP 1.1 虽然也可通过 PipeLine 实现并发请求，但是 PipeLine 是通过串行传输的，多个请求之间的响应可能会被阻塞。

这里我们有必要明确一下 TCP 连接复用和 HTTP 1.1 中 keep-alive 连接复用的区别：TCP 复用传输是发生在传输层的，而 keep-alive 控制的文件的连接复用是在应用层的；keep-alive 的连接复用是串行的，即一个文件传输完后，下个文件才能复用这个连接，而 TCP 复用是帧的多路复用，即不同文件的传输帧可以在一个 TCP 连接中一起同时进行流式传输。

- HTTP 2 支持传输流的优先级和流量控制机制。HTTP 2 中每个文件传输流都有自己的传输优先级，并可以通过服务器来动态改变，服务器会保证优先级高的文件流先传输。例如在未来的浏览器端渲染中，服务器端就可以优先传输 CSS 文件保证页面的渲染，然后在 CSS 文件全部传输完成后加载 JavaScript 脚本文件。其实这就和我们现在前端一些优化规则有点相背离，例如使用 HTTP 2 的情况下 CSS 文件就不一定要写在 HTML

的顶部，JavaScript 也不一定要在 HTML 最底部写了，因为 HTTP 2 的服务器自动就能帮你做这件事情。

- 支持服务器端推送。服务端能够在特定条件下把资源主动推送给客户端。就像浏览器端的资源预加载一样，例如资源推送可以在 HTML 文档下载之前让 HTML 的 JavaScript 或 CSS 文件先进行下载，从而大大缩短页面加载渲染的等待时间。

所以基于这些 HTTP 2 的优势特性，有人提出：如果推广 HTTP 2，原有网站的一些优化规则将不再适用。其实两者也不完全是矛盾的。一方面，使用 HTTP 2 会极大程度上提高网络的传输效率，让我们可以更少地关注页面的性能优化，是对现在开发优化手段的一个增强。另一方面，现有的优化规则依然能对前端资源的加载和执行起到进一步优化的作用。同时，我们也必须要了解，HTTP 2 和我们还有相当一段距离，目前支持 HTTP 2 协议传输的浏览器依然很少，至少需要达到 EDGE 13、Chrome 45 或 Safari 9.2 以上版本。随着技术的发展和浏览器的更新迭代，HTTP 2 的时代终会到来，但我们依然不能在短时间内企图通过它来帮我们进行页面优化。

## 2.2 web安全机制

Web 前端安全方面涵盖的内容较多，也是前端项目开发中必须要关注的一个重要部分。在 Web 站点开发中，如果没有很好的安全防护措施，不仅可能因为攻击者的恶意行为影响站点页面功能、泄露用户授权隐私，甚至还可能会直接带来用户经济上的损失。尽管如此，我们现在依然会找到大量含有 Web 端安全漏洞的页面，那么这一节我们就来理一理与 Web 前端安全方面相关的问题。

### 2.2.1 基础安全知识

XSS（Cross Site Script，跨站脚本攻击）、SQL（Structured Query Language，结构化查询语言）注入和 CSRF（Cross-site Request Forgery，跨站请求伪造）均属于基础的前端安全知识，逐个来看一下。

#### 👉 XSS

XSS 通常是由带有页面可解析内容的数据未经处理直接插入到页面上解析导致的。需要注意的是，XSS 分为存储型 XSS、反射型 XSS、MXSS（也叫 DOM XSS）三种。这里区分不同类型主要是根据攻击脚本的引入位置：存储型 XSS 的攻击脚本常常是由前端提交的数据未经处理直接存储到数据库然后从数据库中读取出来后又直接插入到页面中所导致的；反射型 XSS 可

能是在网页 URL 参数中注入了可解析内容的数据而导致的，如果直接获取 URL 中不合法的并插入页面中则可能出现页面上的 XSS 攻击；MXSS 则是在渲染 DOM 属性时将攻击脚本插入 DOM 属性中被解析而导致的。XSS 主要的防范方法是验证输入到页面上所有内容来源数据是否安全，如果可能含有脚本标签等内容则需要进行必要的转义。具体看下面几个例子。

<!-- 存储型 XSS，后台从数据库中读取数据返回并在前端页面模板中直接渲染导致存储型 XSS，比较好理解，下面是我们要渲染的内容模板 -->

```
<div>{{ content }}</div>
```

<!-- 渲染后输出内容为 -->

```
<div><script>alert();</script></div>
```

<!-- MXSS，页面标签元素属性在前端渲染时含有可解析的标签导致，下面是要渲染的内容 -->

```
<p calss="class-a {{b}}"></p>
```

<!-- 插入恶意脚本内容的输出结果 -->

```
<p calss="class-a "><script>alert();</script><p class="class-b"></p>
```

// 反射型 XSS，例如 Node 服务器端渲染数据，Web 服务器脚本从前端 URL 中获取数据后直接渲染到前端页面导致  
// 反射型 XSS

```
let name = req.query['name'];
```

```
this.body = `<div>${name}</div>`;
```

// 如果 url 里传递的 name 参数值为 '<script>alert();</script>', 则输出为 <div><script>alert();</script></div>, 会导致页面上的 XSS

常见的 XSS 场景主要包含下面这些，基于这些基本的场景，可以有更多不同的情况出现。所以针对这些问题，我们需要做好数据的校验工作，一般的做法是将所有可能包含攻击的内容进行 HTML 字符编码转义，目前的 HTML 字符编码解码就可以如下实现。

// HTML 字符转译编码

```
function htmlEncode(str) {
  let s = '';
  if (str.length == 0) return '';
  s = str.replace(/&/g, '&amp;');
  s = s.replace(/</g, '&lt;');
  s = s.replace(/>/g, '&gt;');
  s = s.replace(/ /g, '&nbsp;');
  s = s.replace(/\'/g, '&#39;');
  s = s.replace(/\"/g, '&quot;');
  s = s.replace(/\n/g, '<br>');
  return s;
}
```

// HTML 字符转译解码

```
function htmlDecode(str) {
```

```

let s = '';
if (str.length == 0) return '';
s = str.replace(/&/g, '&');
s = s.replace(/</g, '<');
s = s.replace(/>/g, '>');
s = s.replace(/&nbsp;/g, ' ');
s = s.replace(/&#39;/g, '\'');
s = s.replace(/&quot;/g, '\"');
s = s.replace(/<br>/g, '\n');
return s;
}

```

这样，`script` 等内容的标签符号就会直接在页面中显示，而不是被浏览器解析成 `script` 的 DOM 节点来执行了。

```
<div>{{ htmlEncode(content) }}</div>
```

```
<!-- 特殊字符转义后，只显示在页面上，不会被解析-->
```

```
<script>alert();</script>
```

## 👉 SQL注入攻击

SQL 注入攻击主要是因为页面提交数据到服务器端后，在服务器端未进行数据验证就将数据直接拼接到 SQL 语句中执行，因此产生执行与预期不同的现象。主要防范措施是对前端网页提交的数据内容进行严格的检查校验。

```

let id = req.query['id'];
let sql = `select * from user_table where id=${id}`;

let data = exec(sql);
this.body = data;

```

例如以上实例，如果前端传入的 `id` 内容为 `"100 or name=%user%"`，那么查询出来的结果就不只是 `id=100` 的用户了，包含 `user` 字符用户名的用户内容也都会被查询出来，并且这些用户信息都可能被输出，导致 SQL 注入的发生。所以这时我们需要对传入的 `id` 内容进行检验，检查是否包含非法内容。

## 👉 CSRF

CSRF 是指非源站点按照源站点的数据请求格式提交非法数据给源站点服务器的一种攻击方法。非源站点在取到用户登录验证信息的情况下，可以直接对源站点的某个数据接口进行提交，如果源站点对该提交请求的数据来源未经验证，该请求可能被成功执行，这其实并不合理。通常比较安全的是通过页面 Token（令牌）提交验证的方式来验证请求是否为源站点页面提交的，来阻止跨站伪请求的发生。

如图 2-6 所示，用户通过源站点页面可以正常访问源站点服务器接口，但是也有可能被钓鱼进入伪站点来访问源服务器，如果伪站点通过第三方或用户信息拼接等方式获取到了用户的信息，直接访问源站点的服务器接口进行关键性操作（例如支付扣款或返回用户隐私信息等操作），此时如果源站点服务器未做校验防护，伪站点的请求操作就可以被成功执行。另一种情况则可能是盗刷源站点的登录等接口来暴力破解用户密码的情况，如果源站点不添加防护措施，用户信息就极可能被盗取，所以我们需要进行安全性验证。

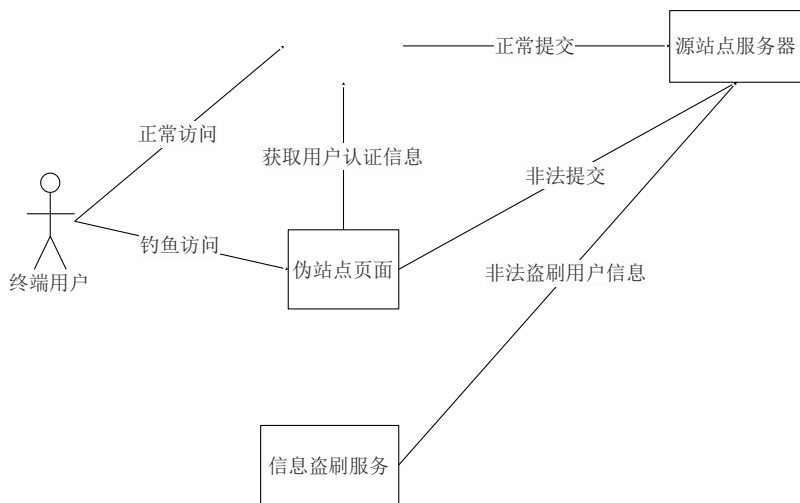


图 2-6 CSRF 攻击原理

如图 2-7 所示，我们在源站点服务请求调用时添加了对源站点的验证，使用服务器端实时返回加密的验证 Token 给源站点页面，在源站点页面提交时将 Token 一起带给服务器验证，而 Token 是不会被其他伪站点利用的。而非法的伪站点和盗刷的行为就可以被直接拒绝掉，这样就大大降低了 CSRF 发生的概率。所以在 Web 后端，我们常常会进行 Token 的验证，其中一种形式是将页面提交到后台的验证 Token 与 session 临时保存的 Token 进行比较就可以实现了。

```
// 生成随机的 csrf 验证 Token，并返回给前端页面
this.session.csrf = md5(Math.random(0, 1).toString()).slice(5, 15);
this.body = yield render('user/login', {
  csrf: ctx.session.csrf
});
```

```
// 提交时验证 Token 是否与源站的 Token 相同
let csrf = this.request.body['csrf'];
if (csrf !== this.session.csrf) {
  res = {
    code: 403,
```



```
    msg: '不明网站来源提交'
  }
} else {
  // 正常提交后的处理逻辑
}
```

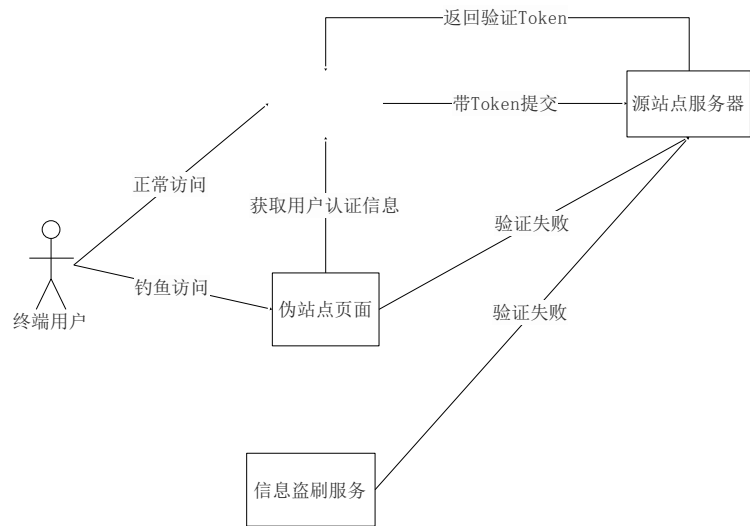


图 2-7 CSRF 预防机制

目前解决 CSRF 的最佳方式就是通过加密计算的 Token 验证,而 Token 除了通过 session 也可以使用 HTTP 请求头中 Authorization 的特定认证字段来传递。当然并不是说使用了 Token,网站调用服务就安全了,单纯的 Token 验证防止 CSRF 的方式理论上也是可以被破解的,例如可以通过域名伪造和拉取源站实时 Token 信息的方式来进行提交。另外,任何所谓的安全都是相对的,只是说理论的破解时间变长了,而不容易被攻击。很多时候要使用多种方法结合的方式来一起增加网站的安全性,可以结合验证码等手段大大减少盗刷网站用户信息的频率等,进一步增强网站内容的安全性。

## 2.2.2 请求劫持与HTTPS

现在除了正常的前后端脚本安全问题,网络请求劫持的发生也越来越频繁。网络劫持一般指网站资源请求在请求过程中因为人为的攻击导致没有加载到预期的资源内容。网络请求劫持目前主要分为两种: DNS 劫持与 HTTP 劫持。下面具体看看两种方式各是什么样的。

👉 DNS劫持

DNS 劫持通常是指攻击者劫持了 DNS 服务器，通过某些手段取得某域名的解析记录控制权，进而修改此域名的解析结果，导致用户对该域名地址的访问由原 IP 地址转入到修改后的指定 IP 地址的现象，其结果就是让正确的网址不能解析或被解析指向另一网站 IP，实现获取用户资料或者破坏原有网站正常服务的目的。DNS 劫持一般通过篡改 DNS 服务器上的域名解析记录，来返回给用户一个错误的 DNS 查询结果实现。

如图 2-8 所示，DNS 劫持症状可能为在某些地区的用户在成功连接宽带网络后，访问域名为 www.a.com 的网站，出现的却是 www.b.com 网站的内容，因为 DNS 服务器 www.a.com 域名的解析结果被修改指向了 www.b.com 网站指向的 IP 地址。

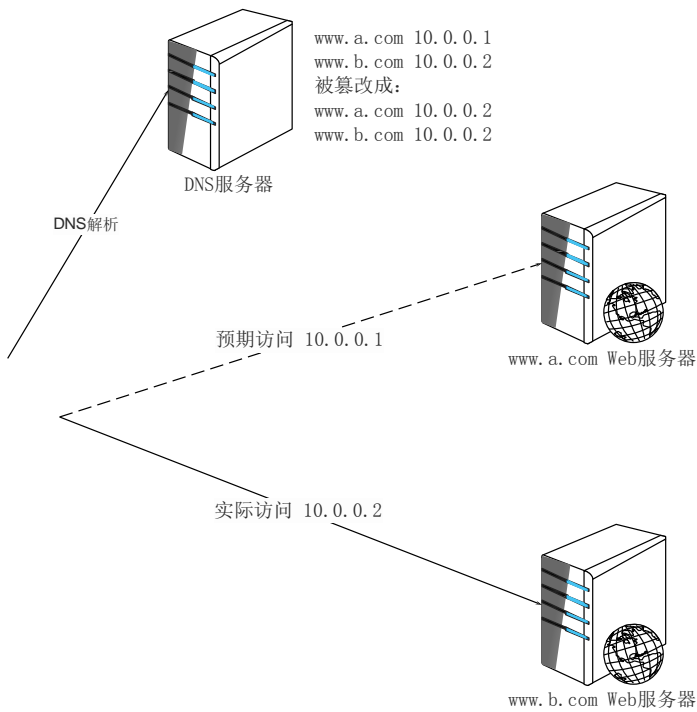


图 2-8 DNS 劫持原理

👉 HTTP劫持

HTTP 劫持是指，在用户浏览器与访问的目的服务器之间所建立的网络数据传输通道中从网关或防火墙层上监视特定数据信息，当满足一定的条件时，就会在正常的数据包中插入或修改成为攻击者设计的网络数据包，目的是让用户浏览器解释“错误”的数据，或者以弹出新窗

口的形式在使用者浏览器界面上展示宣传性广告或者直接显示某块其他的内容。

如图 2-9 所示, 这种情况下一般用户请求源网站的 IP 地址及网站加载的内容和脚本都是正确的, 但是在网站内容请求返回的过程中, 可能被 ISP (Internet Service Provider, 互联网服务提供商) 劫持修改, 最终在浏览器页面上添加显示一些广告等内容信息。

对于这些情况, 网站开发者常常就无法通过修改网站代码程序等手段来进行防范了。请求劫持唯一可行的预防方法就是尽量使用 HTTPS 协议来访问目标网站。

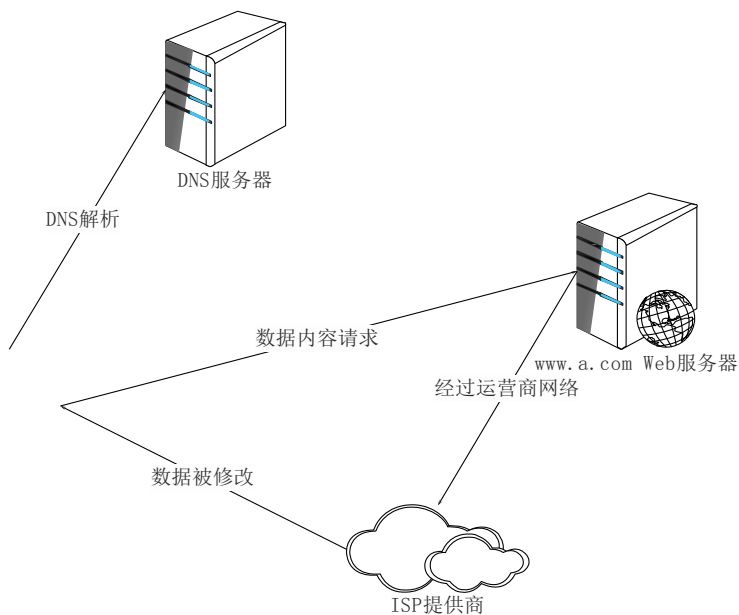


图 2-9 HTTP 请求劫持原理

### 2.2.3 HTTPS协议通信过程

HTTPS 协议是通过加入 SSL (Secure Sockets Layer) 层来加密 HTTP 数据进行安全传输的 HTTP 协议, 同时启用默认的 443 端口进行数据传输。那么使用 HTTPS 是怎样保证浏览器和服务端之间数据安全传输的呢? 我们需要先理解两个概念: 公钥和私钥。

公钥 (Public Key) 与私钥 (Private Key) 是通过一种加密算法得到的密钥对 (即一个公钥和一个与之匹配的私钥), 公钥是密钥对中公开的部分, 私钥则是非公开的部分。公钥通常用于会话加密、验证数字签名或者加密可以用相应私钥解密的数据。通过这种算法得到的密钥对保证是唯一的。使用这个密钥对的时候, 如果用其中一个密钥加密一段数据, 则必须用另一个密

钥解密。比如用公钥加密数据就必须用私钥解密，如果用私钥加密也必须用公钥解密，否则解密将不会成功。我们以公钥加密方式为例，来看看 HTTPS 进行消息安全通信的整个过程。

如图 2-10 所示，客户端在需要使用 HTTPS 请求数据时，首先会发起连接请求，告诉服务器将建立 HTTPS 连接；服务器收到通知后自己生成一个公钥并将它返回给客户端，如果是第一次请求，同时还要告诉客户端需要进行连接验证；如果需要验证，客户端接收到服务器公钥后开始发送验证请求，将一个特定的验证串使用服务器返回的公钥加密后形成密文发送给服务器，同时客户端也将自己生成的公钥发送给服务器；服务器获取到加密的报文和客户端公钥，先使用服务器私钥解密报文获得验证串，然后将验证串通过接收到的客户端公钥加密后返回给客户端；客户端再通过私钥解密验证串，判断是否为自己开始发送的验证串；如果正确，说明双方的连接是安全的，连接验证成功，客户端开始将后面的数据通过服务器初始返回的公钥不断加密发送给服务器，服务器也不断解密获取报文，并通过客户端公钥加密响应的报文内容返回给客户端验证。这样就建立了 HTTPS 双向的加密传输连接。

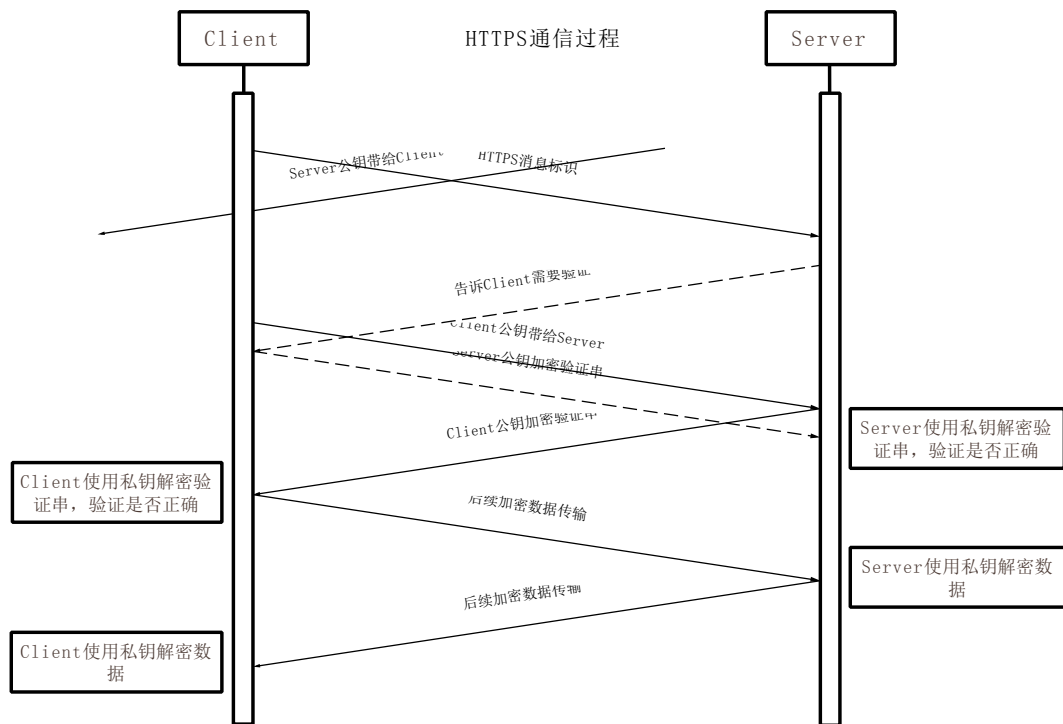


图 2-10 HTTPS 通信建立过程

在这种情况下，传输层传输的内容不会以明文的方式显示，而且 HTTPS 的请求只能被添加

了对应数字证书的应用层代理拦截，因此第三方攻击者就无计可施了。通常我们要创建 HTTPS 服务，在服务端可以使用对应的模块来实现，例如在 Node 端就可以用以下方法来实现。

```
// 引入 https 模块
const httpsModule = require('https');
const fs = require('fs');

// 加载网站 https 服务证书文件，证书一般需要注册申请
const https = httpsModule.Server({
  key: fs.readFileSync('/path/to/server.key'),
  cert: fs.readFileSync('/path/to/server.crt')
}, function(req, res){
  res.writeHead(200);
  res.end("hello world\n");
});

// https 默认监听端口 443
https.listen(443, function(err){
  console.log("https listening on port: 443");
});
```

当然，如果使用 Web 框架，也可以通过更简单的方式创建一个 HTTPS 服务器。

```
const koa = require('koa');
const app = koa();

// 同时监听多个端口
app.listen(80);
app.listen(443);
```

## 2.2.4 HTTPS协议解析

那么，HTTPS 通信和 HTTP 通信方式有什么不同呢？接下来我们具体看看 HTTPS 通信的内容，我们以打开 <https://github.com/ouvens> 页面内容的过程为例，来理解 HTTPS 请求消息的一些关键性结构。

```
Request URL:https://github.com/ouvens
Request Method:GET
Status Code:200 OK (from cache)
Remote Address:192.30.252.131:443
Request Headers
```

图 2-11 所示为 HTTPS 的请求头部内容，图 2-12 所示为 HTTPS 的响应头部内容。从图中可以看出，HTTPS 请求报文和 HTTP 的请求报文区别不大，但是在请求的头部域字段多了 `upgrade-insecure-requests`，该头部字段指令很关键，它可以用于让页面打开的后续请求自动从

HTTP 请求升级到 HTTPS 请求。否则如果使用 HTTPS 来加载 HTML 文件，而 HTML 中加载的是 HTTP 链接的资源文件，则会产生 Mixed Content 类型的错误，并且无法加载资源。考虑到这个问题，W3C 在 2015 年 4 月出台了一个 Upgrade Insecure Requests 的草案，作用就是让浏览器自动升级后面请求为 HTTPS 请求。同时我们在服务器端响应头域中也要加入下面的头域来返回给浏览器，否则浏览器默认安全显示策略会阻塞内容并提示 block-all-mixed-content 类型的错误。

```
header("Content-Security-Policy: upgrade-insecure-requests");
```

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8
Cache-Control: max-age=0
Connection: keep-alive
Cookie: octo=GH1.1.1139190484.1452477402; logged_in=yes; dotcom_user=ouvens; _gh_sess=eyJ3YXN0X3dy
XRRIjjoXDU4MDk2NjU5MDY3LzZzZXNzaW9uX2lkIjo1YzQ0OTdlMmQ2ZDM3MmUyMzhmNT1lMmWI1NDc1Y2IxODgifQ%3
D%3D--d325338462324fd3f46bc103355b644ad4cc0190; user_session=aeWkxGgl3sAnsCvHpTD3B1t2_63jk-Pc2L
Je-9euRmakOXPFvUjKGcnU0cMothpSrKH_sCjMv1znKj5-; _ga=GA1.2.261276230.1452477402; tz=Asia%2FShang
hai
Host: github.com
Referer: https://github.com/
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/4
9.0.2623.87 Safari/537.36
```

图 2-11 HTTPS 请求头部内容

```
Content-Security-Policy: default-src *; base-uri 'self'; block-all-mixed-content; child-src 'self'
render.githubusercontent.com; connect-src 'self' uploads.github.com status.github.com api.githu
b.com www.google-analytics.com github-cloud.s3.amazonaws.com wss://live.github.com; font-src as
sets-cdn.github.com; form-action 'self' github.com gist.github.com; frame-src 'self' render.git
hubusercontent.com; img-src 'self' data: assets-cdn.github.com identicons.github.com www.google
-analytics.com collector.githubapp.com *.gravatar.com *.wp.com *.githubusercontent.com; media-s
rc 'none'; object-src assets-cdn.github.com; plugin-types application/x-shockwave-flash; script
-src assets-cdn.github.com; style-src 'self' 'unsafe-inline' assets-cdn.github.com
Content-Type: text/html; charset=utf-8
Date: Wed, 16 Mar 2016 03:24:32 GMT
Public-Key-Pins: max-age=300; pin-sha256="WoiWRyIOVNa9ihaBciRSC7XHj1YS9VwUGOIud4PB18="; pin-sha2
56="JbQbUG5JMJUoI6brnx0x3vZF6jlXsapgXGVfjhn8Fg="; includeSubDomains
Server: GitHub.com
Set-Cookie: user_session=aeWkxGgl3sAnsCvHpTD3B1t2_63jk-Pc2LJe-9euRmakOXPFvUjR8I9maqWe1lpMDPct1w8
Vdv2FQ0t; path=/; expires=Wed, 30 Mar 2016 03:24:32 -0000; secure; HttpOnly
Status: 200 OK
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
Transfer-Encoding: chunked
Vary: X-PJAX
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-GitHub-Request-Id: 67071C87:680C:361C974:56E8D1EF
X-GitHub-Session-Id: 95501858
X-GitHub-User: ouvens
X-Request-Id: 14ee4ec4fc933b11d957c1233e17edc4
X-Runtime: 0.157269
X-Served-By: 3e68290776c691d4ed02ce5a0c891c6a
X-UA-Compatible: IE=Edge,chrome=1
X-XSS-Protection: 1; mode=block
```

图 2-12 HTTPS 响应头部内容

## 2.2.5 浏览器Web安全控制

除了 HTTPS 外，Web 前端浏览器设置的安全性的控制还有很多，通过某些特定的 head 头配置，就可以完成浏览器端的安全性设置。下面我们再来看几个典型的安全消息头域设置。

### 👉 X-XSS-Protection

这个 head 消息头设置主要是用来防止浏览器中的反射性 XSS 问题的发生，通过这种方式可以在浏览器层面增加前端网页的安全性。不过目前，只有较高版本的 Internet Explorer、Chrome 和 Safari (webkit) 支持这个消息头域字段设置。X-XSS-Protection 通常设置如下。

```
X-XSS-Protection: 1;
mode=block 0 - 关闭对浏览器的 xss 防护; 1 - 开启 xss 防护
mode=block 可以开启 xss 防护并通知浏览器阻止而不是过滤用户注入的 xss 脚本
```

### 👉 Strict-Transport-Security

Strict Transport Security (STS) 是一种用来配置浏览器和服务端之间安全通信的机制，主要用来防止中间者攻击，因为它强制所有的通信都使用 HTTPS，在普通的 HTTP 报文请求中配置 STS 是没有作用的，而且攻击者也能更改这些值。为了防止这样的现象发生，很多浏览器内置了一个配置 STS 的站点列表，在 Chrome 浏览器下可以通过访问 `chrome://net-internals/#hsts` 查看浏览器中站点的 STS 列表，一般 STS 的配置实现如下。

```
max-age=31536000 - 告诉浏览器将域名缓存到 STS 列表中，只有这些特定域名下的资源内容才允许被加载，时间是一年
max-age=31536000;
includeSubDomains;
preload; - 告诉浏览器将域名缓存到 STS 列表里面并且包含所有的子域名，并可支持预加载，时间是一年
max-age= 0 - 告诉浏览器移除在 STS 缓存里的域名，或者不保存当前域名
```

### 👉 Content-Security-Policy

我们简称它为 CSP，这是一种由开发者定义的安全策略性声明，通过 CSP 所约束的规则设定，浏览器只可以加载指定可信的域名来源的内容（这里的内容可以是脚本、图片、iframe、font、style 等等远程资源）。通过 CSP 协定，Web 只能加载指定安全域名下的资源文件，保证运行时的内容总处于一个安全的环境中。

```
Content-Security-Policy:default-src *; base-uri 'self'; block-all-mixed-content;
child-src 'self' render.githubusercontent.com; connect-src 'self' uploads.github.com
status.github.com api.github.com www.google-analytics.com github-cloud.s3.amazonaws.com
wss://live.github.com; font-src assets-cdn.github.com; form-action 'self' github.com
gist.github.com; frame-src 'self' render.githubusercontent.com; img-src 'self' data:
assets-cdn.github.com identicons.github.com www.google-analytics.com
```

```
collector.githubapp.com *.gravatar.com *.wp.com *.githubusercontent.com; media-src  
'none'; object-src assets-cdn.github.com; plugin-types application/x-shockwave-flash;  
script-src assets-cdn.github.com; style-src 'self' 'unsafe-inline' assets-cdn.github.com
```

上面的代码是图 2-2 中 Content-Security-Policy 的配置内容，这里定义了较多的设置，其中 block-all-mixed-content 就是之前提到的，HTTPS 请求的 HTML 会控制阻塞外部 HTTP 资源的文件加载。

### 👉 Access-Control-Allow-Origin

Access-Control-Allow-Origin 是从 Cross Origin Resource Sharing (CORS) 中分离出来的。这个头部设置是决定哪些网站可以访问当前服务器资源的设置，通过定义一个通配符或域名来决定是单一的网站还是所有网站可以访问服务器的资源。需要注意的是，如果服务器端定义了通配符“\*”，那么服务端的 Access-Control-Allow-Credentials（是否允许请求时携带验证信息）选项就无效了，此时用户浏览器中的不同域 Cookie 信息将默认不会在服务器请求里发送（即如果需要实现带 Cookie 进行跨域请求，则要明确地配置允许来源的域，使用任意域的配置是不合法的）。

Access-Control-Allow-Origin : \* \*- 通配符允许任何远程资源来访问 Access-Control-Allow-Origin 的内容

http://www.domain.com - 只允许特定站点才能访问当前资源

Access-Control-Allow-Origin 常常作为跨域共享设置的一种实现方式，其他常用的跨域手段还有：JSONP(JSON with Padding)、script 标签跨域、window.postMessage、修改 document.domain 跨子域、window.name 跨域和 WebSocket 跨域等。

参考资料：

<https://www.w3.org/TR/2014/WD-CSP11-20140211/>。

<http://www.html5rocks.com/en/tutorials/security/transport-layer-security/>。

## 2.3 前端实时协议

在实际的前端应用项目中，除了使用应答模式的 HTTP 协议进行普通网络资源文件的请求加载外，有时也需要建立客户端（这里主要指浏览器）与服务端之间的实时连接进行通信，例如网页实时聊天的应用场景，这就必须涉及浏览器端的实时通信协议了。对于这些对实时性要求较高的应用场景，普通的 HTTP(S) 协议就并不适用。虽然前端可以通过 Ajax 定时向服务端轮询的方式来持续获取服务端的消息，但是这种方式效率相对较低，目前一般只用来处理浏览器上降级体验的实时场景。包括 AJAX 的方式在内，目前可用来在前端浏览器上进行实时通



信的功能实现方式主要有 WebSocket、Poll、Long-poll 和 DDP 协议。

2.3.1 WebSocket通信机制

在本章第一节中我们讲过,HTTP 1.1 的协议支持使用 Upgrade 头域设置进行协议扩展切换,这样就可以实现从 HTTP 1.1 协议切换到其他通信协议进行通信了。幸运的是,现在所有的浏览器基本都支持 HTTP 1.1 协议,一种很典型的实时通信协议便是 WebSocket,WebSocket 是浏览器端和服务器端建立实时连接的一种通信协议,可以在服务器和浏览器端建立类似 Socket 方式的消息通信,关于 WebSocket 的连接过程可以参考图 2-4。

相对于 HTTP 1.1 协议,WebSocket 协议的优势是方便服务器和浏览器之间的双向数据实时通信。但我们要明白的是,HTTP 2 也支持服务端的消息推送,也是可以来适应这一场景的,不过这是未来的事情了。先简单看一下官方 WebSocket 协议中数据帧的定义格式。

图 2-13 为 Websocket 官方参考标准的消息数据帧结构。

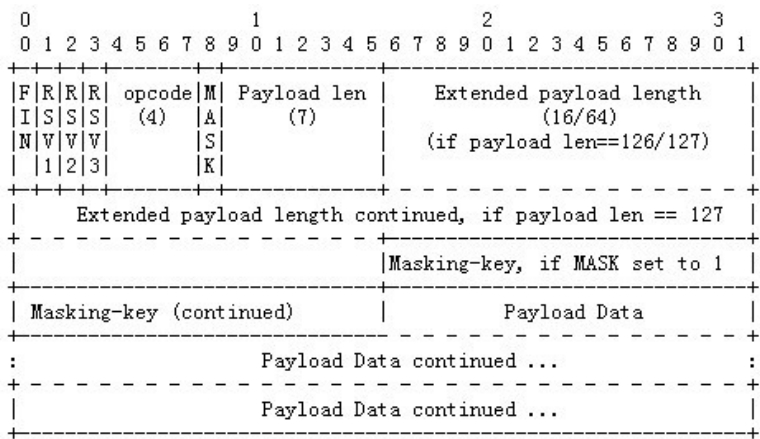


图 2-13 Websocket 数据帧结构

FIN: 1 位。表示当前数据帧是否为消息的最后一帧,一个报文消息由一个或多个数据帧构成,如果消息只由一帧构成,那么起始帧就是结束帧。

RSV1, RSV2, RSV3: 各 1 位。如果未定义扩展,那么这三位都是 0;如果定义了扩展,则为非 0 值;如果接收的帧此处非 0,并且扩展中没有该值的定义,那么关闭连接。

OPCODE: 4 位。表示 PayloadData 有效负荷,如果接收到未知的 OPCODE,接收端必须

关闭连接。

- 0x0 表示附加数据帧；
- 0x1 表示文本数据帧；
- 0x2 表示二进制数据帧；
- 0x3-7 暂时无定义，为以后的非控制帧保留；
- 0x8 表示连接关闭；
- 0x9 表示 ping；
- 0xA 表示 pong，ping、pong 用于保持客户端与服务器之间的心跳连接；
- 0xB-F 暂时无定义，为以后的控制帧保留。

**MASK:** 1 位。用于标识 PayloadData 是否经过掩码处理。如果是 1，Masking-key 域的数据即是掩码密钥，用于解码 PayloadData。

值得注意的是，WebSocket 在网络中传输的最小单位也为帧，数据的传输也可以理解为流式的传输。但 WebSocket 目前在项目中使用时仍然存在一些兼容性问题，它还不支持 IE11 以下或 Android 4.4 版本以下的浏览器。所以在实际项目中，如果要考虑低版本浏览器的使用，我们仍要考虑其他的实现方式。

参考资料：<https://tools.ietf.org/html/rfc6455>。

### 2.3.2 Poll和Long-poll

尽管 HTML5 的 WebSocket 为我们提供了实现前端实时化的方案，提供的 API 也很完备。但不幸的是，并非所有浏览器都支持 WebSocket 协议，在桌面或移动端浏览器应用开发时我们仍不能放心地使用它，这时就必须回到现有的 HTTP 协议上考虑采用 Poll（轮询）和 Long-poll（长轮询）的方案来应对实时通信的场景了。

#### 📌 Poll

Poll 方案很容易理解，即浏览器采用定时向服务器发送请求轮询的方法不断发送或拉取消息。如图 2-14 所示，浏览器每隔一秒向服务器发送一次请求，在一秒内服务器更新的内容在下

一次轮询中将被浏览器拉取返回。所以这种方案相对来说实时性较差，而且没有新消息时依然需要不断轮询，比较消耗系统资源。

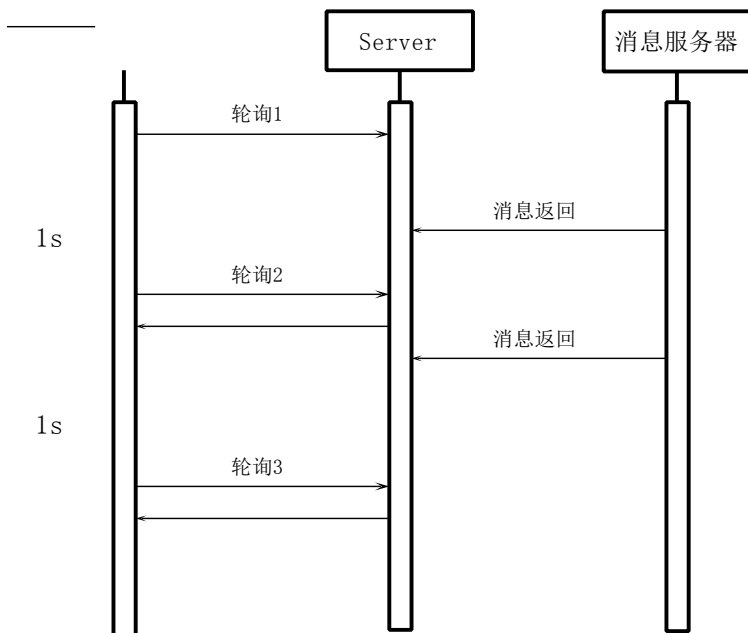


图 2-14 poll 轮询实现原理

### 👉 Long-poll

HTTP 请求可以设置一个较长的 Timeout 等待时间，这样网络轮询请求就可以维持一段较长的时间后返回结果，这也就是 Long-poll（长轮询）的基本思路。服务器只要在这段长轮询时间内进行响应，请求便会立即返回结果；如果这段时间服务器没有返回，浏览器端将自动响应超时并重新发起一个长轮询请求。

如图 2-15 所示，浏览器模拟发起轮询请求后将维持一段时间，这段时间内只要有服务器消息返回，则返回成功，随之立即重新发送一个新的长轮询等待服务器响应。

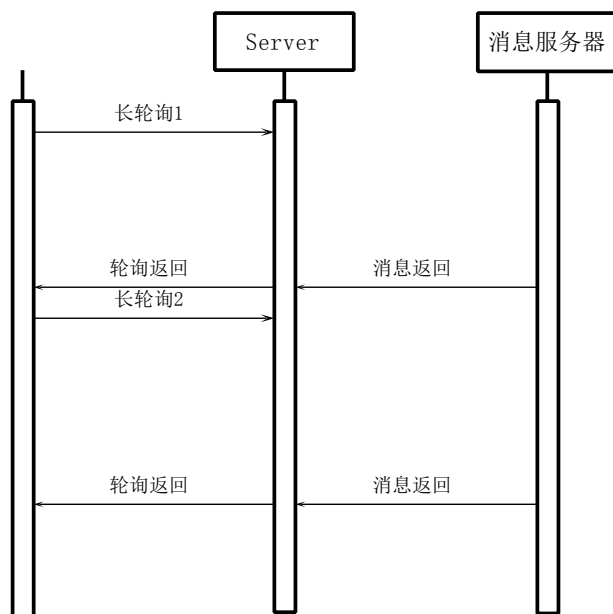


图 2-15 Long-poll 轮询实现原理

相比于 Poll，Long-poll 的实现更加节省系统资源，实时性更好，不用持续地定时发送网络请求。Long-poll 目前一个很典型的应用场景就是网站通过对应的移动客户端进行扫描二维码登录，即用户使用移动客户端扫描二维码登录网站，成功后桌面浏览器页面自动响应跳转进入一个新的登录后页面。

如图 2-16 所示，用户打开桌面浏览器页面后会立即发送一个用户登陆状态查询的长轮询请求，同时开始使用移动客户端扫描二维码，扫描成功时移动客户端会调用接口改变用户的登陆状态，此时服务器可以不断轮询获取用户登录状态改变通知，一旦检测到用户使用移动客户端扫码登录，就将用户登录状态返回给浏览器的长轮询请求，用户浏览器请求到用户登录状态后完成后面的跳转，前端请求登录状态的轮询就可以使用 AJAX 来模拟实现。其实和一般的请求没有太大的差别。

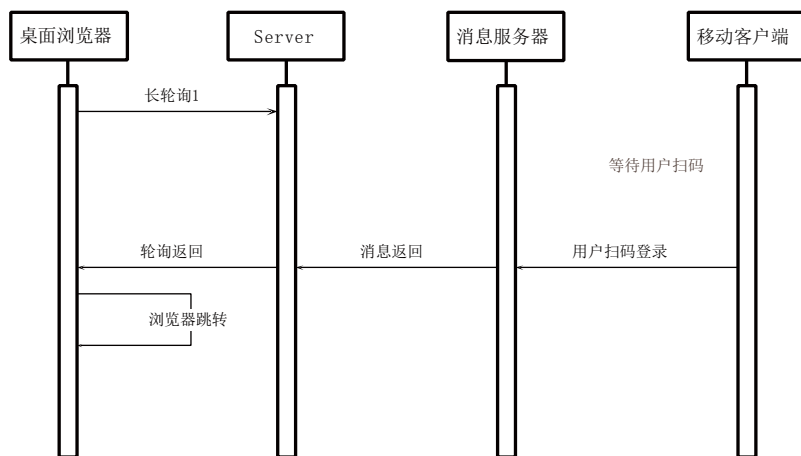


图 2-16 二维码扫描登录跳转原理

```

function _getQrAuth () {
    const self = this;

    // 请求查询登录态
    $.ajax({
        url: '/api/v1/user/qrcode/auth',
        type: 'get',
        dataType: 'json',
        cache: false,
        timeout: 30 * 1000, // 设置 30 秒超时时间
        success: function(data) {

            // 登录成功后自动跳转
            if (data.retcode === 200) {
                window.location.href = '/user_info.html?backUrl=' +
                encodeURIComponent(backUrl);
            }
        },
        error: function(e) {
            console.log(JSON.stringify(e));
        }
    });
}

```

在 Node.js 服务端对应的处理程序可以用如下方法来实现登陆态的循环查询。

```

const qrcodeAuth = function*(req, res) {
    let ctx = this;

    let authState = false;
    while(true){

```

```
// 查询用户登录态，已登录则返回用户信息，否则返回 false
authState = queryAuthState(ctx);
if(authState){
  // 如果用户已登录，则返回成功结果
  ctx.body = {
    retcode: 200
  }
  return ;
}
ctx.body = {
  retcode: 500,
  msg: 'time out'
}
};
```

Web 应用中客户端和服务端建立实时通信的方式比较多，包括 HTML5 提供的 WebSocket、Flash 实现的 WebSocket、XMLHttpRequest 轮询长连接、XMLHttpRequest Multipart Streaming、script 标签的长时间轮询等。不常用的方式暂不介绍过多，大家有兴趣可以自行了解。

### 2.3.3 前端DDP协议

DDP (Distributed Data Protocol, 分布式数据协议) 是一种新型的客户端与服务器端的实时通信协议，由于兼容性的原因，目前使用还不广泛。DDP 使用 JSON 的数据格式在客户端和浏览器之间进行数据传输通信，所以对于前端开发者来说使用非常方便。有名的 Meteor Web 框架的双向实时数据更新机制底层使用的就是 DDP，这种协议模式下客户端可向服务器端发起远程过程调用，客户端也可以订阅服务端数据，在服务端数据变化时，服务器会向客户端发起通知，触发浏览器响应的操作。当然我们借助 DDP 模块也可以创建一个简单的 DDP 协议的服务。

```
// 引入 DDP 模块，创建客户端连接器
const DDPClient = require('ddp');
const client = new DDPClient({
  host: 'localhost',
  port: 3000
});

// 监听消息
client.on('message', function(data, flags) {
  console.log('[DDP 消息]: ', data);
});
```

```
// 连接
client.connect(function() {
  // 客户端订阅 post
  client.subscribe('post', [], function() {
    console.log('[post 订阅消息]');
  });
});
```

以 Meteor 为例，在 Web 浏览器端也需要引入相应的协议解析处理模块来完成相应的通信。

```
<script src="./path/meteor-ddp.js"></script>
```

需要注意的是，在浏览器端使用 DPP 协议仍然存在部分兼容性问题，所以目前 DDP 仍没有被广泛使用在实际项目中，但我们可以认为它是面向未来的一种前端实时协议，以后极有可能被广泛使用。

## 2.4 RESTful数据协议规范

REST (Representational State Transfer, 表述性状态转化) 并不是某一种具体的协议，而是定义了一种网络应用软件之间的架构关系并提出了一套与之对应的网络之间交互调用的规则。与之类似的例如早期的 Webservice，当然 Webservice 现在基本都不用了。而在 REST 形式的软件应用服务（这里讨论的主要是 Web 应用服务）中，每个资源都有一个与之对应的 URI 地址，资源本身都是方法调用的目标，方法列表对所有资源都是一样的，而且这些方法都推荐使用 HTTP 协议的标准方法，例如 GET、POST、PUT、DELETE 等。如果一个网络应用软件的设计是按照 REST 定义的，我们就可以认为它使用的交互调用的方法设计遵循 RESTful 规范。

换种方式理解，RESTful 是一种软件架构之间交互调用数据的协议风格规范，它建议以一种通用的方式来定义和管理数据交互调用接口。为了方便理解，我们来看一个具体场景。一个项目经过了需求阶段和评审阶段后便要着手去开发了，团队的前端工程师和后台工程师需要商量前后台的数据协议该怎样定义。例如，对于书籍 book 的记录管理接口，有增、删、改、查操作，于是我们用 path/addBook、path/deleteBook、path/updateBook、path/getBook 来定义接口看上去好像没有什么问题。后来，另一个项目也有类似的接口定义，却可能叫作 path/appendBook、path/delBook、path/modifyBook、path/getBook。接着有一天，项目负责人可能会说，要升级接口来满足新的需求，于是我们又添加了 path/addBook2、path/deleteBook2、path/updateBook2、path/getBook2。这样用起来是没有什么问题，但是这些随意的定义会增加数据接口维护难度和项目继续开发的成本。

或许对于你来说，使用 add、append 或后面添加 add1、add2、add3 都没有问题，但是如果

想将项目转给其他人继续维护，请尽量不要这样做，因为这样的数据交互协议的制定比较乱，没有统一的规范会很难管理。

这时，我们或许会考虑使用文档或规范，规定一定要使用 `add` 来添加，新的接口版本号放前面 `path/v2/addBook`，开发的人必须严格按照文档规范去写。这样做很好，但依然不够完善，原因有以下几点：一是因为项目工作常常排期紧张，你可能没时间去写文档，或者后面接手的人不想去看文档；二是开发修改功能后很可能来不及或忘记去更新文档；三是无论文档写得多么清楚，我们看起来效率总是很低。这时如果有一个风格更好的通用规范来定义数据交互接口，就不用这么麻烦了。

所以我们完全可以利用 **RESTful** 设计的规范特性来解决上面遇到的问题。对于书籍记录操作接口的命名可以如下操作。

如表 2-1 所示，使用 **RESTful** 规范来重新设计接口后，一切就变得很清晰自然，这样新的工程师接手项目时，只要他足够了解 **RESTful** 规范，则几乎没有时间成本。即使他不了解 **RESTful** 规范，也可以很快地去了解，这就可以避免他去读那份看似完善其实冗长杂的文档。

表 2-1 RESTful 风格接口定义

HTTP 方法	URI	描 述
POST	path/v1/book	新增书籍信息，例如添加新书籍
DELETE	path/v1/book	删除书籍信息，例如移除下架某本书籍的信息
PUT	path/v1/book	全量更新书籍信息，例如修改某本书籍的信息
DISPATCH	path/v1/book	更新书籍部分信息
GET	path/v1/book	获取书籍信息

所以，我们开发时就可以这样来定义 **Web** 端接口路由了，对于前后端交互时，只需要引用资源 **URI** 即可。

```
const bookApi = require('../controller/book');
const router = require('koa-router')();
let bookUri = '/path/v1/book';

// 书籍相关 Api
router.post(bookUri, bookApi.addBook);      // 添加书籍记录
router.get(bookUri, bookApi.selectBook);    // 查询书籍记录
router.delete(bookUri, bookApi.deleteBook); // 删除书籍记录
router.put(bookUri, bookApi.updateBook);    // 修改书籍记录

module.exports = router;
```



如果按照这个格式来定义接口，未来即使接口内容需要升级也会变得很简单，只需要修改 `bookUri` 的路径就可以了。

如表 2-2 所示，我们可以使用某一级路径来清晰地标识接口的版本号，当然更加标准的 RESTful 定义方式可能是将版本号写在 HTTP 请求头信息的 `Accept` 字段中进行区分。但其实这样会给我们的开发带来一些麻烦，例如不能在开发阶段直观地看出这个接口是针对哪个版本功能逻辑的描述，必须打开请求的头部才能得到接口版本，因此推荐使用某一级路径来清晰地标识接口的版本号信息，尽管这和规范在一定程度上是相背离的。

表 2-2 RESTful 风格升级版本接口定义

HTTP 方法	URI	描 述
POST	path/v2/book	新增书籍信息，例如添加新书籍
DELETE	path/v2/book	删除书籍信息，例如移除下架某本书籍的信息
PUT	path/v2/book	全量更新书籍信息，例如修改某本书籍的信息
DISPATCH	path/v2/book	更新书籍部分信息
GET	path/v2/book	获取书籍信息

RESTful API 的主要设计原则就是这些，总结来说就是结合 HTTP 的固有方式来表征资源的状态变化描述，而不是通过动词加名词的方式来设计。这种定义风格可以让数据交互的方式更加规范化，一定程度上有利于降低项目开发和维护的成本。

## 2.5 与Native交互协议

移动互联网兴起后，智能移动设备出现，移动端 Native 开发（我们现在一般将移动端原生应用的开发称为移动端 Native 开发）几乎一夜之间盛行起来，并很快得到第一批掘金者的追捧。相比于之前 Wap Web（可以理解为移动互联网兴起前手机端的网页应用）应用的简单网页开发，Native App 以更优的性能、更好的用户体验以及 Google、Apple 平台厂商对开发者的共赢支持引领了移动互联网时代智能应用软件开发的第一波浪潮。随之而来的就是 HTML5 的出现，它允许开发者在移动设备上快速开发网页端应用，也可以将 Web 页面嵌入到 Native 应用中，它的到来让移动互联网应用开发很快进入到了 Native App、Web App、Hybrid App 并存的时代。随着移动互联网第一波浪潮渐渐过去，Hybrid App 结合了 Native App 和 Web App 的优势，在牺牲一小部分性能的前提下，适应了更多的移动应用开发场景，成为目前广为使用的移动端开发模式。当然 Native App 和 Web App 今天也依然有它们各自适用的应用场景。基于这个背景我们来具体看看 Hybrid App 中与前端相关的协议与应用。

### 2.5.1 Hybrid App应用概述

Hybrid App 是在 Native App 应用的基础上结合了 Web App 应用所形成的模式，我们称之为混合 App。从技术开发上来看，相比于传统的桌面浏览器端的 Web App，它具有以下几方面明显的特征。

- Hybrid App 可用的系统网络资源更少。由于移动设备 CPU、内存、网卡、网络连接多方面的限制，Hybrid App 的前端页面可用的系统资源远远小于桌面浏览器。就网络连接来说，大部分移动设备的使用者使用的仍是 3G、4G 甚至 2G 的网络，带宽和流量均有限制，和桌面浏览器的带宽接入相比还是有着本质上的区别。
- 支持更新的浏览器特性。我们知道目前智能设备浏览器种类相对较少，且随着硬件设备的快速更新，主流的浏览器以 WebKit 内核居多，支持较新的浏览器特性。不像桌面浏览器那样需要考虑较低版本 Internet Explorer 的兼容性问题。
- 可实现离线应用。Hybrid 的一个优势是通过新的浏览器特性或 Native 的文件读取机制进行文件级的文件缓存和离线更新。这是桌面浏览器上较难做到的。这些离线机制常常可以用来弥补 Hybrid App 网络系统资源不足的缺点，让浏览器脚本更快从本地缓存中加载。
- 较多的机型考虑。由于目前移动设备平台的不统一性，而且不同设备机型系统的浏览器实现仍有一定的区别，因此 Hybrid App 应用仍需要考虑不同设备机型的兼容性问题。
- 支持与 Native 交互。Hybrid App 的另一个特点是结合了移动端 Native 特性，可以在前端页面中调用客户端 Native 的能力，例如摄像头、定位、传感器、本地文件访问等。而这些也是我们在本节中要关注的内容。

所以在实际的 Hybrid App 项目开发中尤其需要注意上述问题，针对这些问题，我们需要做好交互、优化和兼容性的工作。其中前端与 Native 的交互是目前 Hybrid 应用实现过程中非常重要的一个环节，也是设计实践过程中比较复杂的一个部分。本节中，我们先来了解 Hybrid App 中前端与 Native 交互方面的知识，看看 Hybrid App 与 Native 之间的交互协议的具体内容。

### 2.5.2 Web到Native协议调用

在 HTML5 中调用 Native 程序一般有两种较通用的方法，下面逐一来看。

## 通过URI请求

首先来看一下 Hybrid App 中如何通过 URI 请求在 HTML5 前端页面中来调用一个 Native 的方法或界面。其主要原理是，Native 应用可在移动端系统中注册一个 Scheme 协议的 URI，这个 URI 可在系统的任意地方授权访问来调起一段原生方法或一个原生的界面。同样，Native 的 WebView 控件中的 JavaScript 脚本的请求也可以匹配调用这一通用的 Scheme 协议。例如我们通过对 `window.location.href` 赋值或使用 `iframe` 的方式发送一个 URI 的请求，这个请求可以被 Native 应用的系统捕获并调起 Native 应用注册匹配的这个 Scheme 协议内容。代码如下。

```
let iframe = document.createElement('iframe');
iframe.setAttribute('style', 'display:none');
document.body.appendChild(iframe);
iframe.setAttribute('src', 'myApp://className/method?args');
```

如图 2-17 所示，一般 Native 应用会提前向移动端系统注册 Scheme 协议，之后前端脚本发送 `iframe` 中的 URI 资源请求时，WebView 会将获取 URI 资源的地址交给 Native App，Native App 将请求转发给系统并进行解析。此时如果你的 Native 应用注册了与之匹配的 Scheme URI 到系统，系统就会通过此 URI 地址执行该 Scheme 协议定义的 Native 操作，执行一段 Native 代码或者拉起 App 的某个界面（例如打开摄像头、打开移动 App 首页等）。这样就完成了 HTML5 中 JavaScript 对 Native 的调用。

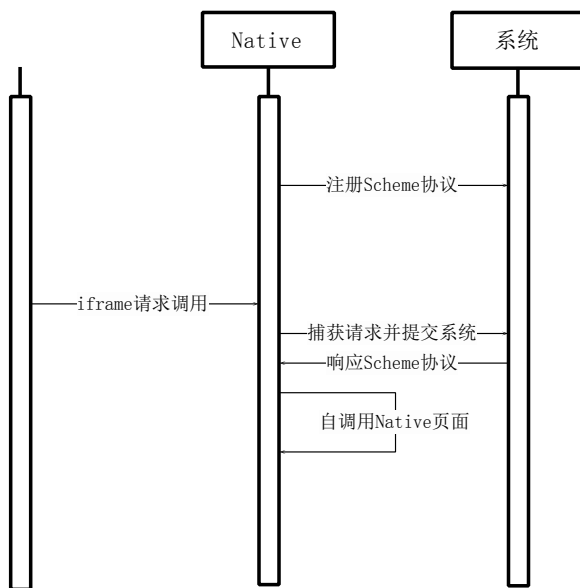


图 2-17 Scheme 协议注册调用过程

## 通过addJavascriptInterface注入方法到页面中调用

除了使用 URI 方式调用 Native 注册的 Scheme 协议以外，还可以通过 addJavascriptInterface 方法将 Native 的一个对象方法注入到页面中，供 JavaScript 调用。这里就以一个 Android 平台的例子来看下具体是如何使用的。

```
// 声明 webSettings 的实例
WebSettings webSettings = webView.getSettings();
// 设置 webView 内可执行 JavaScript 脚本
webSettings.setJavaScriptEnabled(true);
// 加载页面到 WebView 中
webView.loadUrl("file:///android_asset/index.html");
// 添加 native 类实例注入到 webView 中
webView.addJavascriptInterface(new JsInterface(), "native");
public class JsInterface {
    @JavascriptInterface
    public void showToast(String toast) {
        Toast.makeText(MainActivity.this, toast, Toast.LENGTH_SHORT).show();
    }
}
```

这里，Native 会向 webView 的全局作用域中注入一个 native 的全局对象，并且 native 对象中具有 showToast 方法，而这个方法是可以通过前端页面中 JavaScript 调用的。对应的 HTML5 页面则可以按如下方式编写。

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>demo</title>
</head>
<body>
    <script>
        nativeAlert('hello ouven');

        // 这里就可以调用 native 实例对象的 showToast 方法了
        function nativeAlert(msg){
            native.showToast(msg);
        }
    </script>
</body>
</html>
```

通过向 WebView 的全局作用域注入 native 对象的方法，WebView 中页面的 window 对象就可以直接通过前端的 native.showToast 方法调用 Native 应用中 JsInterface 的动作了。当然这里是

个很简单的实例，它的主要实现原理是通过 `addJavascriptInterface` 将 Java 的实例对象注入到 `WebView` 中，让 `WebView` 中的页面 JavaScript 可以直接使用，采用这种方法也可以实现更加复杂的调用功能。

### 2.5.3 Native到Web协议调用

相反，如果 Native 需要主动调用 HTML5 页面中 JavaScript 方法或指令，又该怎么做呢？同样地，也需要先使用 JavaScript 在 HTML5 页面全局中声明相对应的方法，这就有点类似于 Native 注册的 Scheme 协议。其中，Native 向 HTML5 发起的调用是通过 `loadUrl` 方法（Android 平台下面方法名为 `loadUrl`，iOS 系统下通常为 `stringByEvaluatingJavaScriptFromString`）实现的，例如使用 `webView.loadUrl("javascript: alert('hello ouven')");` 则可以执行 HTML5 页面的 `alert` 方法。再来看一个稍微复杂一点例子。

```
// 声明 webSettings 的实例
WebSettings webSettings = webView.getSettings();
// 设置 WebView 内可执行 JavaScript 脚本
webSettings.setJavaScriptEnabled(true);
// 加载页面带 WebView 中
webView.loadUrl("file:///android_asset/index.html");
// 声明 JsInterface 实例
JsInterface jsInterface = new JsInterface();
jsInterface.log('hello ouven');

public class JsInterface {

    public void log(final String msg){
        webView.post(new Runnable() {
            @Override
            public void run() {
                // log 是 js 的 log 方法
                webView.loadUrl("javascript: log(" + "'" + msg + "'" + ")");
            }
        });
    }
}
```

这段 Java 代码中，Native 希望通过创建一个新线程来执行页面 JavaScript 的 `log()` 方法，相对应的 HTML5 页面可按如下方式实现在全局作用域声明一个 `log()` 方法供 Native 调用。

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<meta charset="UTF-8">
<title>demo</title>
</head>
<body>
  <script>
    // 在页面中声明方法
    function log(msg){
      console.log(msg);
    }
  </script>
</body>
</html>
```

采用这种方法便可以实现 Native 中调用 JsInterface 实例中的 `log()` 方法时控制 WebView 在页面中打印信息。相信对大家来说应该很容易理解。

总结来看，这里实现交互的核心方法其实都可以认为是通过方法注入来实现的，Native 应用将协议注入到系统 Scheme，或将 Native 方法直接注入到页面的全局变量，反之也可以在 HTML5 页面全局作用域中添加方法，让 Native App 调用。这样就完成了前端与 Native App 的相互调用。

## 2.5.4 JSBridge设计规范

了解 Android 的人应该知道，使用 `addJavaScriptInterface` 在 Android 4.2 以下版本会有一些安全漏洞。例如，通过 JavaScript 可以访问当前设备 SD 卡上面的任何内容，甚至是联系人信息、短信等。虽然 Android 4.2 以上的版本可以通过添加 `@JavascriptInterface` 的方法来解决安全隐患，但是对于 Android 4.2 以下的版本就得考虑其他的实现方式了。幸运的是，在 Android 上，JavaScript 还可以通过另一种 `setWebChromeClient` 方法来实现：JavaScript 在执行 WebView 中 JavaScript 的 `alert()` 或 `prompt()` 方法时，Native 端会自动触发 `onJsAlert` 或 `onJsPrompt` 的方法回调函数，一般情况下，因为前端需要常常使用 `alert()` 方法，因此通过重写 Native 中的 `onJsPrompt` 方法，JavaScript 就可以通过执行 `prompt()` 方法把数据内容传递到 Java 代码中执行，即 JavaScript 在执行 `prompt()` 方法时将数据传入到 Native 的 `onJsPrompt` 方法中，而 `onJsPrompt` 里则是我们重写的调用 Native 程序代码。

```
// 设置 prompt 监听
webView.setWebChromeClient(new WebChromeClient() {
    @Override
    public boolean onJsPrompt(WebView view, String url, String message, String defaultValue,
JsPromptResult result) {
        result.confirm(JSBridge.callJsPrompt(MainActivity.this, view, message));
        return true;
    }
});
```

```

    }
  });

```

同时 Java 也可以主动通过 `loadUrl()` 再次回调 JavaScript 的方法返回内容到 `WebView` 中。具体需要 JavaScript 将 Native 调用所需要方法的名称、参数和 Native 执行完成后回调 JavaScript 的方法名称等信息都传给 Native。

以 Android 为例，目前使用较多的解决方案是通过一个协议串定义 Native 和 JavaScript 间的数据通信规则：`jsbridge://className:callbackMethod/methodName?jsonObj`，当然不一定需要这样的定义，也可以使用其他方式，这只是一种方法。无论如何，这个协议串必须包括以下内容：调用 Native App 的特定标识头、类名称、方法名、参数、回调 JavaScript 的方法。这里，`jsbridge` 是 Native 注册的协议头，`className` 对应的是调用 Native 的类，`methodName` 是指调用 Native 类中具体哪个方法，`jsonObj` 则是指 JavaScript 调用 Native 方法时传入的参数，`callbackMethod` 为回调 JavaScript 的方法名称，目的是在 JavaScript 调用 Native 的方法成功后异步通过 `loadUrl('javascript: callbackMethod()')` 来让 JavaScript 继续进行操作。

如图 2-18 所示，以 Android 为例，我们需要在前端调用 Native 层 `Util` 类的 `toString` 方法，传入参数为 `msg`，调用成功后执行前端 JavaScript 的 `success` 方法，传入的协议地址代码如下。

```
jsbridge://Util:success/toString?{"msg":"hello ouven"}
```

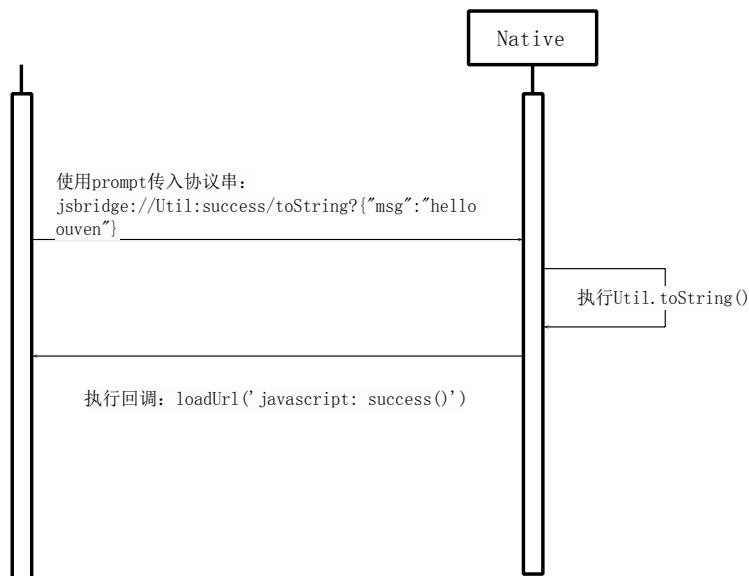


图 2-18 JSBridge 协议调用方式

这时，Native 应用就需要实现 `Util` 类的 `toString` 方法了。当 Native 应用接收到

`jsbridge://Util:success/toString?{"msg":"hello ouven"}` 协议串时便调用 `Util.toString()` 方法，完成后还要调用 `loadUrl('javascript:success()')` 方法执行前端的回调。

声明注册较多的类和方法需要花费较多的精力而且较难管理，所以我们通常希望可以在 `WebView` 端使用一个通用注册接口来注册我们可能需要使用到的所有类和方法，例如 `JSBridge.register("jsName", javaClass.class)`，注册完成后通过某个 `JavaScript` 全局对象来管理查看，以供 `Native` 端使用 `loadUrl()` 方法来调用。

```
JSBridge.call = function(className, methodName, params, callback) {
    let bridgeString;
    let paramsString = JSON.stringify(params || {});

    // 拼接协议串
    if (className && methodName) {
        bridgeString = `jsbridge://${className}:${callback}
/${methodName}?${paramsString}`;
        try {
            // 将协议串发送给 Native 应用
            sendToNative(bridgeString);
        } catch (e) {
            console.log(e);
        }
    } else {
        console.log('Invalid className or methodName');
    }
}

// 发送协议串到 Native
function sendToNative(uri, data) {
    window.prompt(uri, JSON.stringify(data || {}));
}
```

此处，`JavaScript` 执行时也使用类似上面的 `JSBridge.call(className, methodName, params, callback)` 来调用对应的方法。`JSBridge.call()` 方法做的事情很简单，将传入的参数组装成上述 `jsbridge://className:callbackMethod/methodName?jsonObj` 的形式，然后调用 `prompt` (`iOS` 一般用 `iframe`，`Android` 建议使用 `window.prompt`) 的方法将协议串传递到 `Native` 应用层解析执行，这时候 `Native` 层会收到这个协议串，再进一步解析并调用对应的类和方法名称，调用方法成功后执行 `WebView` 中声明的 `callbackMethod()` 函数，这样整个交互协议调用的过程就完成了。通过这种方式，我们就定义了前端与 `Native` 的相互调用协议。



## 2.6 本章小结

本章主要为大家介绍了与前端相关的协议，包括 HTTP 1.x、HTTP 2、HTTPS、安全协议、WebSocket、Poll、Long-poll、RESTful 协议规范以及在 Hybrid 应用中前端与 Native 交互协议的设计。下一章中，我们将正式探讨 Web 相关的前端技术，主要从前端三层结构及其演进发展来展开介绍，让大家对前端基础技术的发展有系统的把握。

# 第 3 章

## 前端三层结构与应用

相信大家已经了解了前端的三个基本构成：结构层 HTML、表现层 CSS 和行为层 JavaScript。

结构层 HTML 现在已经发展到 HTML5 版本，而在实际工程项目中，HTML5 一般只有在移动端页面开发中才会使用到，桌面浏览器页面开发时由于兼容性原因，使用的虽然是 HTML5 的 doctype 定义但一般不使用 HTML5 的新标签规范。幸运的是，HTML5 标准是向后兼容的，HTML4 版本的绝大部分常用标签依然可以继续使用。

对于表现层 CSS，我们从 CSS2 开始了解就可以了。目前 CSS3 已经成熟，同样在移动端浏览器开发时使用，不过桌面浏览器开发时也会使用已支持的部分 CSS3 属性。另外，CSS4 的草案正在制定当中，但距离发布仍有一段时间。

JavaScript 标准也在 ECMAScript 5 发布后经历了几年的波折才确定发布了新版本。ECMAScript 6 于 2015 年 6 月 17 日发布，ECMAScript 7 标准也于 2016 年完成发布，目前 ECMAScript 6+（泛指 ECMAScript 6 及以上版本）已经成为新的 JavaScript 标准规范正在被广泛使用。同时，这一标准的确定也让现代前端的三层结构有了更加明确的新定义。

此外我们也需要了解的是，前端三层结构是基础。现代的 Web 前端应用开发早已经不是简单的三层结构就能轻松解决的了，而是已经形成了编译流程化、生产环境基础优化结构运行的模式。简单来讲，例如 HTML 开发可以由 Component（实现的形式较多，例如 Web Component、目录级 Component、其他框架自定义形式的 Component）来管理结构，CSS 由 SASS、postCSS、stylus 等预处理器的语法开发来代替，JavaScript 则使用 ECMAScript 6+、TypeScript 等特性标准进行高效开发。这些就是目前主要的前端开发技术，其主要过程是开发完成后将开发项目管理的三层结构内容编译输出为浏览器支持运行的基础三层结构解释执行。为什么会有这么多复杂的东西出现呢？主要是对效率的需求！通过更高效的工具来快速开发和管理复杂的应用项目，

最后编译为基础结构运行是因为仍然有旧的浏览器需要兼容。如果某一天所有浏览器都直接支持 Component、ECMAScript 6+、SASS 等，那就进入了浏览器直接解释运行的时代，不需要自己再转译了。

自前端技术诞生以来，就一直保持着较快的发展速度。同时，前端技术的快速发展对前端工程师的要求也越来越高。无论怎样，因为对效率需求迫切，现代前端的编译开发技术已经成为了主流。这一章，我们将重点来学习现代前端三层结构的演进历程以及如何在三层结构的基础之上进行高效开发。

## 3.1 HTML结构层基础

### 3.1.1 必须要知道的DOCTYPE

提到 DOCTYPE，我们不妨先从 HTML 4.01 开始讲起。HTML 4.01 是 W3C 在 1999 年制定发布的 HTML 语言规范。要知道的是，HTML4.01 是基于 SGML（Standard Generalized Markup language，标准通用标记语言）规范来制定的。HTML5 正式发布于 2014 年，HTML5 不是基于 SGML 演化而来的，可以理解是为 W3C 的另一套实现规范。HTML5 向后兼容了绝大多数低版本的 HTML 元素标签，并添加了新的元素标签，例如<header>、<nav>、<footer>、<section>、<video>、<audio>等。

另外，虽然目前几乎所有浏览器均支持以 HTML5 的方式声明文档类型，即<!DOCTYPE html>，但并不代表 HTML5 的新标签元素就可以在这些浏览器上正常解析，这是因为 DOCTYPE 声明只用于指示 Web 浏览器页面使用哪个 HTML 版本编写的指令进行解析，<!DOCTYPE html>的定义兼容所有 HTML 的历史版本和最新的 HTML5 版本，不支持 HTML5 中的 DOCTYPE 定义的浏览器仍然会使用 HTML 4.01 等历史版本的兼容模式来进行文档解析。

SGML 的规范很多，所以我们以前使用 HTML 开发页面时需要为文档类型定义 (Document Type Definition, DTD) 进行声明，即在 DOCTYPE 后面声明 DTD 的定义，不同的文档 DTD 会指示浏览器以不同的文档模式来解析 HTML 文本。如果 DOCTYPE 不存在或格式不正确，则会导致文档以兼容模式呈现，这时浏览器会使用较低的浏览器标准模式来解析整个 HTML 文本。如果定义了标准模式运行，例如<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">或<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/

`strict.dtd">`,那么这时浏览器 DOM 渲染和 JavaScript 的运作模式都是以该浏览器支持的最高标准来解析执行的。

HTML 最早从 XML 衍生而来,目前已经进入 HTML5 成熟阶段。要了解的是,HTML5 不是基于 SGML 的,不需要对 DTD 进行定义。实际上我们可能早已不再使用旧版本的声明模式了,但还是有必要了解其中的原因。

### 3.1.2 Web语义化标签

首先给出 Web 语义化标准的定义:Web 语义化是指在 HTML 结构的恰当位置上使用语义恰当的标签,使页面具有良好的结构,使页面标签元素具有含义,能够让人或搜索引擎更容易理解。

基于此定义,我们需要首先理解以下几点。

- 用正确的标签做正确的事情。例如,在列表的地方要使用列表元素,文章内容中要使用段落或文章标签,杜绝 HTML 结构全部使用<div>元素来嵌套实现,因为<div>是没有任何语义的,仅代表一个元素容器块。

```
<!-- 不推荐 -->
<div class="ui-menu-list">
  <div class="menu-list-item">列表一</div>
  <div class="menu-list-item">列表二</div>
  <div class="menu-list-item">列表三</div>
</div>

<!-- 推荐 -->
<ul class="ui-menu-list">
  <li class="menu-list-item">列表一</li>
  <li class="menu-list-item">列表二</li>
  <li class="menu-list-item">列表三</li>
</ul>
```

- HTML 语义化能让页面内容更具结构化且更加清晰,便于浏览器和搜索引擎进行解析,因此在兼容条件下,要尽量使用带有语义化结构标签。例如,HTML5 中定义<header>、<nav>、<footer>、<section>、<article>等标签的内容会被搜索引擎解析收录,能够提高页面内容关键字的搜索排行。

图 3-1 为 HTML5 语义化结构标签的一个典型的应用场景,目前使用也很广泛。根据标签就能理解元素里面的大致内容功能,这样自然比所有的元素都用<div>来写更容易理解。

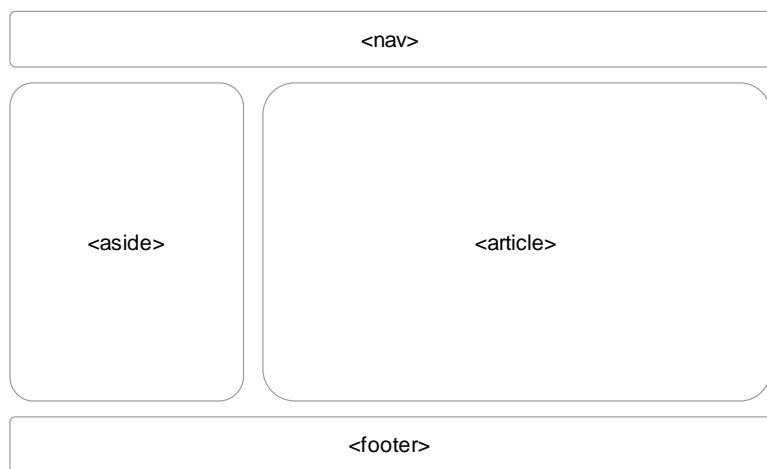


图 3-1 HTML5 部分语义化结构元素标签

```
<!-- 不推荐div布局方式 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>页面结构</title>
</head>
<body>
  <div id="header">
    页面头部
  </div>
  <div id="main">
    正文内容
  </div>
  <div id="footer">
    页面底部
  </div>
</body>
</html>

<!-- 推荐语义化结构 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>页面结构</title>
</head>
```

```
<body>
  <header>
    页面头部
  </header>
  <article>
    正文内容
  </article>
  <footer>
    页面底部
  </footer>
</body>
</html>
```

- 即使在没有样式 CSS 的情况下，网页内容也应该是有顺序的文档格式显示，并且是容易阅读的。

一般情况下，具有良好 Web 语义化的页面结构在没有样式文件的情况下也是能够阅读的，例如列表会以列表的样式展现，标题文字会加粗，而不是全部内容都以无层次的文本内容形式呈现。如图 3-2 所示，左边是使用语义化标签实现的页面结构，而右边则是全部使用 div 布局实现的页面结构，显然，使用语义化标签的页面结构在没有样式的情况下更容易理解。

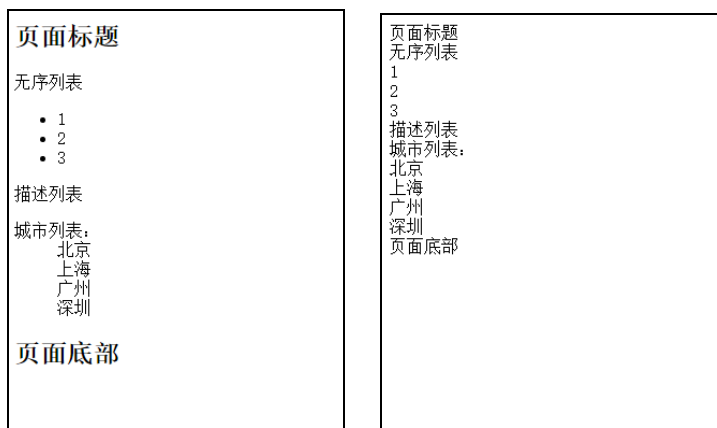


图 3-2 语义化结构页面与非语义化结构页面对比

- 使项目维护人员更容易对网站进行分块，便于阅读理解。语义化的标签使用，能让开发者更容易区分标签元素中的内容，例如，<nav>一般是导航菜单的结构，<article>则用于存放页面的文章内容，更加易于后期的维护，而全部使用<div>来实现就不是那么容易区分了。

理解了以上几点，我们再来看看 HTML 主要标签的设计。CSS 规范规定：每个标签元素都

是有 `display` 属性的。所以根据标签元素的 `display` 属性特点,可以将 HTML 标签分为以下几类。

- 行内元素: 包括`<a>`、`<b>`、`<span>`、`<img>`、`<input>`、`<button>`、`<select>`、`<strong>`等标签元素,其默认宽度是由内容宽度决定的。
- 块级元素: 包括`<div>`、`<ul>`、`<ol>`、`<li>`、`<dl>`、`<dt>`、`<dd>`、`<h1>`、`<h2>`、`<h3>`、`<h4>`、`<h5>`、`<h6>`、`<p>`、`<table>`等标签元素,其默认宽度为父元素的 100%。
- 常见空元素: 例如`<br>`、`<hr>`、`<link>`、`<meta>`、`<area>`、`<base>`、`<col>`、`<command>`、`<embed>`、`<keygen>`、`<param>`、`<source>`、`<track>`等不能显示内容甚至不会在页面中出现,但是对页面的解析有着其他重要作用的元素。

这些标签大部分是有具体含义的,具有其适用的场景。例如, `h1~h6` 是表示标题的标签元素, `ul`、`ol`、`dl` 分别表示无序、有序、带标题描述的列表。而且 HTML5 中加入的新标签也基本是带语义化的。

除此之外,合理的标签使用能够让搜索引擎更容易获取页面的主要内容,提升权重。例如,页面中`<h1>`标题元素里面的文字就比`<div>`标签元素里面的文字更重要,会更容易被搜索引擎记录下来,并认为是对页面描述更有用的内容(这个在后面章节中会具体讲解)。所以我们在设计 HTML 结构时要尽量注意语义化的使用。

### 3.1.3 HTML糟糕的部分

或许没有人告诉过你,Web 语义化规范并不是在任何时候都需要严格遵守的,有时直接使用甚至会产生一些副作用。例如在桌面浏览器应用开发时,若页面标签结构中使用了 HTML5 的新语义化标签可能会导致这些标签无法直接解析,所以考虑到兼容性,我们最终仍然需要使用`<div>`来代替,另外我们或许也会考虑使用 `html5.js` 来进行特殊解析,但是这样在部分版本的浏览器中,页面的解析就变得较慢了。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>页面结构</title>
  <!--[if IE]>
    <script src="http://html5shiv.googlecode.com/svn/trunk/html5.js"></script>
  <![endif]-->
</head>
```

```
<body>
  <header>
    页面头部
  </header>
  <article>
    正文内容
  </article>
  <footer>
    页面底部
  </footer>
</body>
</html>
```

关于前端标签元素或特性的兼容性，我们一般可以通过访问 <http://caniuse.com/> 来查询。

再如页面中使用<table>这个语义化标签是会导致内容渲染较慢的，因为<table>里面的内容渲染是等表格内容全部解析完生成渲染树后一次性渲染到页面上的，如果表格内容较多，就可能产生渲染过程较慢的问题，因此我们常常需要通过其他的方式来模拟<table>元素，例如使用无序列表来模拟表格。

```
<section class="ui-table">
  <ul class="ui-table-list row">
    <li class="table-cell table-title">单元格标题一</li>
    <li class="table-cell table-title">单元格二</li>
    <li class="table-cell table-title">单元格三</li>
  </ul>

  <ul class="ui-table-list row">
    <li class="table-cell">单元格一</li>
    <li class="table-cell">单元格二</li>
    <li class="table-cell">单元格三</li>
  </ul>

  <ul class="ui-table-list row">
    <li class="table-cell">单元格一</li>
    <li class="table-cell">单元格二</li>
    <li class="table-cell">单元格三</li>
  </ul>
</section>
```

除了<table>元素的性能问题，HTML 中还有一些糟糕的设计需要我们注意，例如你可以任意编写HTML嵌套结构，并能够在任何位置使用任何元素，也可以通过内联CSS来改变HTML的任何样式属性，甚至HTML标签随意写或者CSS属性使用错误都不会报错。这样就比较糟糕了，增加了很多犯错的可能性。



所以我们编写 HTML 时尤其需要注意这些问题，下面来看几个常见的 HTML 标签使用的糟糕场景。

```
<!-- <img>元素标签可以不用写 alt 或 title，也能正常显示 -->
```

```

```

```
<!-- <a>元素标签可以不写 href 属性，不过这样容易出现问题。即使添加块级元素也不会报错，但是里面的内容在浏览器解析后会发生位置偏移，如果出了问题将很难定位 -->
```

```
<a><h2></h2></a>
```

```
<!-- 并不是所有的标签都是带有语义化的，<div>、<i>就是比较典型的例子，所以尽量避免在这些标签里面直接添加文字，实际项目开发中，我们常常把<i>元素标签当作页面上的 icon 图标签来使用 -->
```

```
<div></div>
```

```
<i></i>
```

```
<!-- 尽管 HTML 规范提供了有语义化的列表元素，但我们仍然可以用下面这种方式来定义列表，而且在页面上也可以正常显示 -->
```

```
<div>
```

```
  <span class="list-item">1</span>
```

```
  <span class="list-item">2</span>
```

```
  <span class="list-item">3</span>
```

```
</div>
```

```
<!-- 元素中可以使用内联样式，当然这是旧版本的历史问题；元素样式里面随意添加 top 属性也是可以的，只是不生效且不会报错；加入 display: relative; 也不会提示错误，但 relative 并不是 display 的属性 -->
```

```
<div style="width:100px;height:30px;top:10px;display: relative;"></div>
```

```
<!-- HTML 定义了 table 元素，但 table 是一次性渲染的，如果表格内容较长就比较慢了 -->
```

```
<table>
```

```
  <thead>table list</thead>
```

```
  <tr>
```

```
    <th>list 1</th>
```

```
    <th>list 2</th>
```

```
    <th>list 3</th>
```

```
  </tr>
```

```
  <tr>
```

```
    <td>list 1</td>
```

```
    <td>list 2</td>
```

```
    <td>list 3</td>
```

```
  </tr>
```

```
  ...
```

```
  <tr>
```

```
    <td>list n</td>
```

```
    <td>list n+1</td>
```

```
    <td>list n+2</td>
```

```
  </tr>
```

```
</table>
```

<!-- 表单输入项内容不写 label 也是没问题的, <label>可以定义与表单控件间的关系, 当用户选择该标签时, 浏览器会自动将焦点转到和标签相关的表单控件上。-->

Date:<input type="text" name="name" />

<!-- 还有一些很不合理的标签, 新的标准已经将它们弃用了 -->

<blink></blink>

<marquee></maquee>

<stike></strike>

<img>标签的 alt 属性和 title 属性是有区别的, alt 属性一般表示图片加载失败时提示的文字内容, title 属性则指鼠标放到元素上时显示的提示文字。在页面结构书写中, 我们常用 title 来提示一些省略掉的文字的全部内容。例如<p title="这是一段很长的文字, 包含很多内容">这是一段很长的文字…</p>, 这样在页面中可能只展示部分文字, 但用户可以通过鼠标提示看到这段文字的完整内容, 这对于提高页面用户体验是很有帮助的。

很显然, 这些糟糕的设计不仅降低了页面可读性, 拖慢了页面性能, 不利于 SEO, 而且误导了初学者对 HTML 的理解使用, 更有可能让我们在已经出错的情况下找不到错误的原因和方向。因此, 我们必须想办法尽可能避免这些问题的发生。其中的一种思路是, 借助现代开发工具插件或构建工具插件来分析上面 HTML 页面中编码糟糕的地方, 例如发现内容里有行内元素里嵌套了块级元素时, 就通过插件分析报错来辅助我们进行正确的书写开发。辅助工具可以提示一部分问题, 但并非所有的问题都能得到解决。另一方面, 如果我们想让 HTML 的结构进一步优化提升, 可以考虑尝试 AMP HTML。

### 3.1.4 AMP HTML

流动网页提速 (Accelerated Mobile Pages, AMP) 是 google 推行的一个提升页面资源载入效率的 HTML 提议规范。基本思路有两点: 使用严格受限的高效 HTML 标签以及使用静态网页缓存技术来提高网络访问静态资源的性能和用户体验。也就是说, 尽量避免使用目前网页上渲染或展示性能比较差的标签, 并将部分网页静态内容缓存到页面上进行分发, 例如内联体积较小的样式和图片、延时加载较大的静态资源文件等, 进而提高网页的初始载入速度。

AMP 提议中包含了一部分网页端优化的内容, 前端网页优化我们都已经做过了, 在此着重来看 AMP 的第一部分。使用受限的 HTML 标签来进行 AMP HTML 提升, 这只是一个规范提议, 不涉及任何一项具体的技术, 例如在 AMP 中, <img>、<video>、<audio>、<embed>、<form>、<table>、<frame>、<object>、<iframe>这类较慢或可能影响页面内容渲染的标签是不建议被直接使用的, 因为它们常常在页面元素解析时就要去做较慢的渲染或者会立即

直接下载 `src` 或 `param` 等属性里面的内容。`<video>` 标签载入解析时会去直接请求 `src` 里面的资源内容,这样就占用了浏览器的下载线程,阻塞了页面关键资源的下载,所以我们可以将这些元素需要加载的资源先缓存到 HTML 结构中,等页面主体结构渲染完成后再去加载,类似懒加载,不同的是 AMP HTML 是通过自定义元素完成 Component 来实现的,懒加载则是通过 JavaScript 直接在网页中操作实现。

浏览器同一个域名的最大并行下载线程个数是有限的,所以我们常常要先加载页面的关键性展示资源,延后加载页面脚本类资源或页面的非关键性图片资源。一般浏览器 (IE8 以上) 对同一个域名下的资源最多支持 4~6 个并行下载数,所以为了增大资源下载并行数,我们常常将 HTML、JavaScript、CSS、图片资源分域存放。分域也可以将静态资源请求进行服务器端的负载均衡,并对请求中的 cookie 信息进行隔离,因为跨域请求默认是不带 Cookie 的,这样便减小了 JavaScript、CSS、图片等资源的请求头部信息大小,从而提升了请求的解析速度。

```
<amp-video width="400" height="300" src="http://www.domain.com/videos/myvideo.mp4"
poster="path/poster.jpg">
  <div fallback>
    <p>Your browser doesn't support HTML5 video</p>
  </div>
  <source type="video/mp4" src="myvideo.mp4">
  <source type="video/webm" src="myvideo.webm">
</amp-video>
```

这里是 AMP 标签 `<amp-video>` 的一个例子,AMP HTML 提出了一个可选的实现方案就是通过添加自定义元素来代替 AMP 规范限制的元素,即使用 `<amp-video>`、`<amp-img>`、`<amp-audio>`、`<amp-pixel>` 等来做页面元素内容的延迟载入或渲染,其实本质上这类标签的逻辑封装实现和异步加载有点类似。同样,JavaScript 的脚本使用也被严格限制或做了类似的处理,通过完全异步加载的方式来改变和优化页面上资源的加载顺序,保证整体页面内容能尽快完成加载渲染。根据 AMP 团队测试,通过 AMP 规范优化后的网页载入速度可以提升 15%~85%,那么在海量用户请求访问的情况下,这就很有价值了。

但这点似乎和 HTML 设计的初衷相背离,也不是 Web 语义化所提倡的,另外 HTML 不同元素在设计时本身效率是各不相同的,如果让效率较高的元素来代替其他慢元素的高频率渲染,这样 AMP HTML 的优势就可以体现出来了。总结来看,使用 AMP 提升页面性能的基本的原则如下。

- 只允许异步的 script 脚本

- 只加载静态的资源
- 不能让内容阻塞渲染
- 不在关键路径中加载第三方 JavaScript
- 所有的 CSS 必须内联
- 字体使用声明必须高效
- 最小化样式声明
- 只运行 GPU 加速的动画
- 处理好资源加载顺序问题
- 页面必须立即加载
- 提升 AMP 元素性能

以下是一个最简单的 AMP HTML 例子。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <link rel="canonical" href="hello-world.html">
  <meta name="viewport" content="width=device-width,minimum-scale=1,initial-scale=1">
  <style amp-boilerplate>
  body {
    -webkit-animation: -amp-start 8s steps(1, end) 0s 1 normal both;
    animation: -amp-start 8s steps(1, end) 0s 1 normal both;
  }

  @-webkit-keyframes -amp-start {
    from {
      visibility: hidden;
    }
    to {
      visibility: visible;
    }
  }

  @keyframes -amp-start {
    from {
      visibility: hidden;
    }
```

```

    }
    to {
        visibility: visible;
    }
}
</style>
<noscript>
    <style amp-boilerplate>
        body {
            -webkit-animation: none;
            -moz-animation: none;
            -ms-animation: none;
            animation: none;
        }
    </style>
</noscript>
<script async src="https://cdn.ampproject.org/v0.js"></script>
</head>
<body>Hello World!</body>
</html>

```

这里所说的“快”其实并不是单纯指页面渲染速度快，还包括将可以异步或延后的渲染操作延后，在保证用户体验不降低的情况下尽早展示关键性内容。

上面这段代码使用了页面动画的加速，通过 `noscript` 和 `async` 等异步的方式载入 CSS 和 JavaScript 内容，让 HTML 尽可能快地完成渲染解析。其实 AMP HTML 标签的实现原理可以理解为采用自定义的快加载标签元素（标签资源内容延后加载的元素）来代替慢加载标签元素（标签资源内容立即加载的元素）。举个例子，假如网页中的元素分两类：快元素<F>（fast 元素）和慢元素<S>（slow 元素）。<F>类元素比<S>类元素的解析执行时间短 80%（这是完全有可能的），即平均一个<S>元素要用 4 个<F>元素代替。如果现在页面上<F>和<S>类元素数量各为 50%，要完全使用<F>类元素代替，那么页面载入效率提升比例为  $1 - (50\% + 50\% * 4(1 - 80\%)) = 10\%$ ，这是一个很明显的提升，而<F>类元素则可能是 AMP HTML 定义的<amp-table>的实现，用来解决<table>元素慢的问题。显然这种思路在可行性和 AMP 团队的实践结果上面都是很有意义的，而且已经在 Google、Facebook 网站上优化推行了。

而如果需要在实际项目中去推行也比较简单，按照 AMP 标签规范去开发业务组件（也就是 AMP HTML Component）就可以了。所以一个简单的场景，例如我们常用列表元素来代替<table>内容或者图片的懒加载实现，某种意义上就和 AMP 的思想是一致的。

参考资料：<https://www.ampproject.org/>。

## 3.2 前端结构层演进

### 3.2.1 XML与HTML简述

可扩展标记语言（Extensible Markup Language，XML）是用来描述网络上存储数据的一种特殊文本标记格式。它是在 SGML 的基础上演化而来的，XML 推荐使用一对闭合标签的形式来描述数据的名称，这对闭合标签里面的内容则表示对应的值，同时 XML 规定任何一个开始的标签必须与一个结束标签配对，否则将不能按照正确的 XML 结构解析，而且标签之间可以嵌套。我们通过一段 XML 声明便可以使用标签嵌套来书写描述信息了，代码如下。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<info>
  <name>George</name>
  <job>engineer</job>
  <address>
    <country>China</country>
    <city>Shenzhen</city>
  </address>
  <website>http://www.jixianqianduan.com</website>
</info>
```

HTML 则是从 SGML 的基础上演化而来的另一种文本标记语言，一般用于网络上数据的展示。相比而言，XML 更偏向于存储数据，例如我们熟知的 SQLite 数据库，其底层数据就是通过 XML 存储在硬盘上的。而 HTML 则偏重于将数据读取出来装载到浏览器中展示给用户。从语法规则的定义上来看，XML 与 HTML 之间没有必然的联系，它们是针对不同的应用场景而设计的。从语法使用上来看，XML 的灵活度更高，我们可以自定义任意名称的标签来使用 XML，而 HTML 则是具有一定规范约束的，例如只有特定的 HTML 标签才能被浏览器识别。

前面讲过，HTML 4.01 是 W3C 在 1999 年制定发布的 HTML 语言规范，而 HTML5 到 2014 年才正式发布，在这期间我们可以认为页面的结构主要是通过<div>来实现的，曾经我们甚至一度认为前端页面技术就是 CSS + DIV 技术，当然这是在前端技术发展早期，HTML 的元素主要分为行内、块级和常见空元素几类。

HTML5 是 HTML 的第 5 个版本，但它不是基于 SGML 语言演化而来，而是 W3C 完全自定义的一套标准规范。它向后兼容了低版本的 HTML 4.01 的绝大多数标签，并添加了更多语义化标签，例如<header>、<nav>、<footer>、<section>、<video>、<audio>等。需要知道的是，尽管 HTML5 包含了绝大多数的低版本 HTML 标签，但是在浏览器解析时的

<DOCTYPE html>声明中是不需要文档类型定义的。目前遵循 W3C 标准的浏览器均默认支持使用 HTML5 的 DOCTYPE 定义解析，如果不支持则默认使用低版本兼容模式来解析。

### 3.2.2 HTML5 标准

HTML5 增加了较多新的语义化标签，同时对应的 BOM 对象和 DOM 对象也添加了相应的 API，目前由于移动端浏览器内核具有相对较好的统一性和兼容性，所以 HTML5 主要用于移动端页面的开发。我们先来看一下 HTML5 带来的有语义化的元素标签。根据 HTML5 新规范添加的标签，总结起来可以分为以下几类，如表 3-1 所示。

表 3-1 HTML5 新增元素类型

HTML5	HTML4	描述与其他类似标签
<header></header>	<div id="header"></div>	一般用来定义页面的头部模块元素。类似的还有 footer（底部）、nav（导航元素）、section（区段）、hgroup（与 section 组合）等页面结构标签
<video src= "movie.ogg"> </video>	<object type="video/ogg" data="movie.ogv"><param name="src" value="movie.ogv"></object>	用来插入一个视频播放器界面。类似的有 audio（音频）、embed（插件）、canvas（自定义图形，HTML5 之前用 svg）等显示媒体的标签
<source />	<param name="" value="" />	插入视频或音频的属性配置参数标签。
<article></article>	<p></p>	描述一个段落文章元素标签。类似的有 aside（文章外内容）、details（元素细节）、figure（描述信息）等内容类标签。
<time></time>	<span></span>	输出时间标签，类似的有 output（页面输出内容）、mark（页面需要突出的文字）等标记类标签。
<datalist></datalist>	Combox	可选数据的列表，和 input 配合使用。
<progress></progress>	none	显示 JavaScript 等执行进度标签
<command/>	none	定义命令按钮。类似的有 figcaption（定义 figure 元素的标题）、keygen（页面密钥）、meter（度量衡）、summary（描述）、ruby（同 rp、rt 注释 ruby 代码使用）等标签

我们也可以用 HTML5 之前版本的标签来代替。当然，这里也包含一些看似比较实用，实际在项目中并不会经常使用的语义化标签，例如 `<command>`、`<meter>` 等。

HTML5 除了新增一部分元素标签外，还在一部分原有标签中增加了一些新属性。例如 `<input>` 这个元素标签的属性值变化就比较大，`<input>` 新增的属性有 `autocomplete`、`placeholder`、`autofocus`、`required`，另外 `type` 属性也增加了 `email`、`url`、`number`、`range`、`color`、`search`、`date` 等这些类型来满足更多的应用场景，下面的代码就可以实现原生的日期时间选择。

```
<input type="text" autocomplete="on" placeholder="请输入姓名" autofocus required/>
<input type="color" />
<input type="date" />
<input type="time" />
```

用较新的浏览器解析这段 HTML，可以看到这两个标签的定义会出现一个日期时间选择器和一个颜色选择器，如图 3-3 所示。



图 3-3 HTML5 日期时间与颜色选择

不知道你有没有想过，为什么 `<input>` 这么简单的标签定义能生成这样两个较复杂的选择输入界面呢？我们为什么还要常常找第三方日期时间选择器来自己实现呢？类似的标签还有很多，例如 `<video>`、`<audio>` 等。既然是 HTML5 自带的，浏览器原生就应该支持，但浏览器为什么就可以这样做呢？接着来了解下一节的内容，找到这些问题的答案。

### 3.2.3 HTML Web Component

要解决上一节中的问题，必须提到 Shadow DOM。先不急于解释 Shadow DOM 是什么，我们来看一个例子。

```
<video src="./assets/media.mp4" controls autoplay name="media"></video>
```

如图 3-4 所示，`<video>` 这样一个标签可以在浏览器产生一个界面相对复杂且带有播放控



制的播放器，这是如何做到的呢？为了便于理解这个问题，我们以 Chrome 浏览器为例，选择浏览器调试工具设置中的 show userAgent Shadow DOM，就可以看到 Shadow DOM 里的内容。



图 3-4 video 标签浏览器显示效果

```
<video src="./assets/midea.mp4" controls autoplay name="media">
  #Shadow root
  <div pseudo="-webkit-media-controls">
    <div pseudo="-webkit-media-controls-overlay-enclosure">
      <input type="button" style="display: none;">
    </div>
    <div pseudo="-webkit-media-controls-enclosure">
      <div pseudo="-webkit-media-controls-panel" style="display: none;">
        <input type="button" pseudo="-webkit-media-controls-play-button">
        <input type="range" step="any" pseudo="-webkit-media-controls-timeline"
max="0">
        <div pseudo="-webkit-media-controls-current-time-display" style="display:
none;">0:00</div>
        <div pseudo="-webkit-media-controls-time-remaining-display">0:00</div>
        <input type="button" pseudo="-webkit-media-controls-mute-button">
        <input
          type="range"
          step="any"
          max="1"
pseudo="-webkit-media-controls-volume-slider" style="display: none;">
        <input
          type="button"
pseudo="-webkit-media-controls-toggle-closed-captions-button" style="display: none;">
        <input type="button" style="display: none;">
        <input type="button" pseudo="-webkit-media-controls-fullscreen-button"
```

```
style="display: none;">
    </div>
  </div>
</div>
</video>
```

很明显，这里的<video>标签中其实有很多的内容，隐藏的 Shadow-root 里面的内容就是以上视频播放器控制组件 HTML 结构的所在之处，对应的 CSS 也是可以看到的，可见标签内部也是由很多个<div>、<input>和与之对应的 CSS 样式形成的。另外，浏览器之所以将其置灰，是为了表明这部分是 Shadow DOM 里面的内容，在页面中，其他部分是不能调用这里的隐藏元素的。也就是说，你自己写的 CSS 选择器和 JavaScript 代码都不会影响到 Shadow DOM 里面的内容。实质上就是让<video>标签里的逻辑和样式都被浏览器单独封装并与外界元素独立，而<video>标签内容在浏览器上的渲染则是在浏览器结构 UI 后端模块中设置的。

再来具体看下什么是 Shadow DOM，Shadow DOM 是 HTML 的一个规范，它允许浏览器开发者封装自己的 HTML 标签、CSS 样式和特定的 JavaScript 代码，同时也可以让开发人员创建类似<video>这样的自定义一级标签，创建这些新标签内容和相关的 API 被称为 Web Component。这里提出了 Shadow DOM 和 Web Component 两个概念，大家要弄明白两者之间的区别和关联。

Shadow root 是 Shadow DOM 的根节点；Shadow tree 为这个 Shadow DOM 包含的节点子树结构，例如<div>和<input>等；Shadow host 则称为 Shadow DOM 的容器元素，即上面的标签<video>。那么我们该如何使用 Web Component 创建一个 Shadow DOM 呢？

新版本的浏览器提供了创建 Shadow DOM 的 API，指定一个元素，然后可以使用 document.createShadowRoot() 方法创建一个 Shadow root，在 Shadow root 上可以任意通过 DOM 的基本操作 API 添加任意的 Shadow tree，同时指定样式和处理的逻辑，并将自己的 API 暴露出来。完成创建后需要通过 document.registerElement() 在文档中注册元素，这样 Shadow DOM 的创建就完成了。需要注意的是，目前该方法仅支持 chrome 31 及 Android 4.4 以上版本的浏览器。以下为一个图文组合的 Shadow DOM 元素功能实现代码。

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>图文组合插件</title>
  <!-- import 引入组件内容 -->
  <link href="./image.html" rel="import" />
</head>
<body>
```

```

<!-- 这里注册生成了一个 shadow host 为<x-image>的 DOM 元素 -->
<x-image src="./image.jpg" width="290" height="160" alt="带文字描述的图片"></x-image>
</body>
</html>

```

组件模板的代码文件如下。

```

<!-- 定义组件 -->
<template>
  <!-- 组件模板 -->
  <style>
    /* Shadow host 内容 */
    :host {
      display: block;
    }
    :host .x-image {
      position: relative;
      display: block;
      margin: 0;
      padding: 0;
      width: 100%;
    }
    :host .x-image .x-image-image{
      margin: 0;
      padding: 0;
      width: 100%;
      height: 100%;
    }
    :host .x-image .x-image-text{
      position: absolute;
      display: block;
      bottom: 0;
      padding: 10px 5px;
      left: 0;
      right: 0;
      color: #fff;
      background: rgba(0,0,0,0.5);
    }
  </style>

  <div class="x-image ">
    <span class="x-image-image">
      <img class='image' src="" alt="image" height="200">
    </span>
    <span class="x-image-text">demo</span>
  </div>
</template>
<script>
  // 在本文件被导入后自动执行

```

```
(function(doc) {
  "use strict"; // 启用严格模式
  // 所有实例元素对象的公共 prototype
  let XImage = Object.create(HTMLElement.prototype, {
    height: {
      get: function() { return this._height; },
      set: function(height) {
        this._height = height;
        this._innerBanner.style.height = height + 'px';
        this._innerBanner.querySelector('.image').style.height = height + 'px';
      }
    },
    width: {
      get: function() { return this._width; },
      set: function(width) {
        this._width = width;
        this._innerBanner.style.width = width + 'px';
        this._innerBanner.querySelector('.image').style.width = width + 'px';
      }
    },
    alt: {
      get: function() { return this._width; },
      set: function(alt) {
        this._alt = alt;

        this._innerBanner.querySelector('.banner-text').innerHTML = alt;
        this._innerBanner.querySelector('.image').setAttribute('alt', alt);
      }
    },
    src: {
      get: function() { return this._src; },
      set: function(src) {
        this._src = src;
        this._innerBanner.querySelector('.image').setAttribute('src', src);
      }
    }
  });
  // 组件被创建时执行，相当于构造函数
  XImage.createdCallback = function() {
    // 创建 Shadow root，将自定义模板放入其中
    let shadowRoot = this.createShadowRoot();
    let template = doc.querySelector("template");
    let node = document.importNode(template.content, true);

    this._innerBanner = node.querySelector(".banner-section");
    // 设置传入属性，并应用到 Shadow DOM 内容中
    let height = this._height || Number(this.getAttribute("height")),
```

```

        width = this._width || Number(this.getAttribute("width")),
        alt = this._alt || String(this.getAttribute('alt')),
        src = this._src || String(this.getAttribute('src'));

        if (!isNaN(height) || !isNaN(width)) {
            this.height = height;
            this.width = width;
        }
        if(alt){
            this.alt = alt;
        }
        this.src = src;
        shadowRoot.appendChild(node);
    };
    // 注册组件
    document.registerElement("x-image", { prototype: XImage });

})(document.currentScript.ownerDocument);
</script>

```

这里在 HTML 中使用 HTML import 方式引入外部的 Shadow DOM 内容, Shadow host 为 `<x-image>`, 在支持 Shadow DOM 的浏览器上会显示如图 3-5 所示的效果, 即描述文字在图片底部区域的一个悬浮效果, 同时在自定义的组件里我们也可以按照自己的需要向外暴露可配置的属性和 Web API 接口。



图 3-5 带悬浮文字的图片标签显示效果

根据该原理实现 Shadow DOM 的流程就是 Web Component 技术。我们已经知道了应该怎么做, 那么再来看一下利用 Web Component API 创建实现一个 Shadow DOM 时需要注意哪些方面。

- 注册的 Shadow host 名称不能与浏览器自带的原生 Shadow host 名称相同, 这个应该很容易理解, 就是不允许出现两个名称相同但是功能不同的元素标签, 例如 HTML5 规范中含有 `<header>` 元素, 就不能再创建 Shadow host 为 `<header>` 的 Shadow DOM。

```

<!-- 不推荐 -->
<audio>...</audio>
<video>...</video>

```

```
<!--回过头看下 AMP HTML Component，其创建方式也是 Web Component -->
<amp-audio>...</amp-audio>
<amp-video>...</amp-video>
```

- 注意样式模块的隔离，一定要保证当前的 Shadow DOM 样式只在当前 Shadow DOM 内生效。由于我们自己也可以创建 Shadow DOM 的样式，所以开发的时候就要注意规范，否则会造成 Shadow DOM 之间样式耦合、相互影响的问题。一般我们将 Shadow DOM 唯一的 Shadow host 作为类名来定义，例如<x-image>的样式定义可以编写如下。

```
<style>
  /* Shadow host 内容 */
  :host {
    display: block;
  }
  :host .x-image {
    position: relative;
    display: block;
    margin: 0;
    padding: 0;
    width: 100%;
  }
</style>
```

- 暴露合理的有效属性配置和 DOM API 接口。例如上面实例中内容的 src、width、height 属性配置对于元素的最终表现是有影响的，我们在创建 Shadow DOM 的同时也要考虑到 Shadow DOM 的可定制和扩展性。
- 在目前浏览器不完全支持 Web Component 的情况下，如果需要在实际项目中使用，我们还需要解决好如何构建打包的问题。通过构建来分别打包 Component 的 HTML、JavaScript、CSS 可以实现当前主流浏览器的兼容，这种思路实质上就是通过构建解析和创建 Shadow DOM 内容来解析 Shadow DOM 结构，而不是让浏览器直接解析 Shadow DOM 结构。

### 3.3 浏览器脚本演进历史

我们知道，目前前端浏览器上的执行脚本以 JavaScript 为主。但是这并不意味着我们在任何情况下都可以直接使用 JavaScript 进行安全开发，之前有人提出 JavaScript 设计中糟糕的部分，还有人读过《JavaScript 语言精髓》。作为一种标准化的语言，竟然有很多特性被人公然地摒弃，十分不可思议。例如对于 ECMAScript 5 及以前的 JavaScript 版本，虽然已经可以定义使用严格模式了，但我们仍会听到一些不好的评价，例如全局作用域、eval、with、类声明缺失等不合理

的特性设计，后面我们也会具体讨论这些问题。

```
name = 'ouven';
person = {
  name: name,
  address: 'China',
  job: 'engineer',
  city: 'Shenzhen',
  website: 'http://www.jixianqianduan.com'
};

with(person){
  console.log(name, address, job, city, website);
  eval('alert(name)');
}
```

上面的代码在非严格模式的下是可以完美运行的，但是并不合理，全局变量会让 `window` 很难管理，`with` 会降低代码执行速度、`eval` 的使用也有性能缺陷。尽管如此，在不同的时代背景下我们仍需要合理的解决方案来处理和避免这些问题以保证开发的安全性。幸运的是，我们总能找到一些解决方案来应对当前的这些问题，先来看看曾经盛极一时的 `CoffeeScript`。

### 3.3.1 CoffeeScript时代

如今已经很少有人再提起 `CoffeeScript`，但 `CoffeeScript` 代表着前端脚本语言历史上的一段辉煌时期。`CoffeeScript` 是一套可转译为 `JavaScript` 语法的语言。那么，既然已经有了 `JavaScript`，为什么还会有 `CoffeeScript` 的出现？早在 `JavaScript` 规范混乱的时代，一些糟糕的设计不仅让开发的代码运行不稳定，而且还会增加大型项目的维护难度。像前面提到的全局变量、作用域 `this`、函数参数为数组对象、类声明缺失、默认模式下无规范限制、语法声明冗繁等特性在开发过程中尤其需要注意，一旦处理不完善，就可能导致出错并花费大量的时间去排查。

因此 `CoffeeScript` 的创建者 `Jeremy Ashkenas` 借鉴了部分其他语言简洁、开发高效的特性，重新定义了一套语法规则，然后按照统一的规则转译成规范、可读、默认在严格模式下运行的 `JavaScript` 代码，这样就有效地保证了最后运行的 `JavaScript` 代码是具有一定规范且风格统一的，而 `CoffeeScript` 本身定义的语法规则也是十分高效且带有较好设计规范的。尽管这样运行时需要增加转译的工作，但是能帮助我们规避上面这些不安全的问题，是很有价值的。

下面来看一些 `CoffeeScript` 特性，具体如下。

```
#赋值：
number=1
opposite=true
```

```
#条件:
number=-1 if opposite

#函数:
square=(x)->x*x

#数组:
list=[1,2,3]

#存在性:
alert"I knew it!" if isKnown?
```

现在 CoffeeScript 也可以直接转译为 ECMAScript 6 (后面将会具体介绍 ECMAScript 6 的特性) 代码, 转译后代码如下。

```
'use strict';

let number = 1;
let oppsite = true;

if(opposite){
  number = -1;
}

function square (x){
  return x*x;
}

let list = [1,2,3];

if(isKnown){
  alert("I knew it!");
}
```

相比之下, CoffeeScript 语法更加简洁易用, 限制避免了 eval、with 这类不安全语法的使用。CoffeeScript 定义的这套语法转译规则使用了更加简洁高效的编码语法来转译生成严谨安全的 JavaScript, 因此也受到很多 JavaScript 开发者的青睐。甚至有人曾提出要做另一个 JVM (JavaScript Virtual Machine, JavaScript 虚拟机), 希望用它在浏览器上直接运行 CoffeeScript, 可见 CoffeeScript 曾经是多么的盛极一时。笔者也曾维护过一个基于 CoffeeScript 的项目, 整个项目完全使用 CoffeeScript 实现, 路由地址配置代码就多达 7000 多行, 所以 CoffeeScript 在只有 ECMAScript 5 的时代是极具代表性的。

了解 ECMAScript 6 标准的人肯定知道上面例子中的箭头函数。随着 ECMAScript 6 标准的发布, 更加标准的规范和同样类似的高效特性出现了, 此外, ECMAScript 6 标准还补充完善并



增强了 JavaScript 本身的运行特性。CoffeeScript 因此开始走向没落。

### 3.3.2 ECMAScript 标准概述

首先我们来了解一下 ECMAScript，ECMAScript 是 TC39（Technical Committee 39，技术专家委员会 39）负责制定的 JavaScript 标准，其发展历史如下。

- 1999 年 12 月，ECMAScript 3 发布。其中主要定义了 ECMAScript 的一些常用对象和条件控制语法，例如内置对象、正则表达式、条件控制语法、异常处理、错误定义等。
- 2009 年 12 月，ECMAScript 5 标准发布。其中主要新增了严格模式、JSON 对象、新增 Object 接口、新增 Array 接口、Function.prototype.bind 等特性。
- 2015 年 6 月 17 日，ECMAScript 6 正式发布。其中增加了块级作用域变量声明规范、String 模板、解构、arrow 函数、Symbol 类型、类、迭代器、生成器、集合、Promise、Proxy 等特性。这也是 CoffeeScript 开始走向没落的一个转折点。
- 2016 年，ECMAScript 7 发布，主要增加了幂函数和 Array.prototype.includes 特性。

就兼容性方面而言，Node 6.0 版本已基本支持到 ECMAScript 6（93% 特性支持），新版本的 Node 也支持更多 ECMAScript 6 以上的新特性。浏览器端 Chrome 52 已完全支持 ECMAScript 6，并逐渐添加了对 ECMAScript 7 特性的支持，其他浏览器或平台也在陆续添加对 ECMAScript 6 标准的支持。

在工程开发实践中，ECMAScript 6 的应用也很快被推广并分为两个方向：一是用于浏览器端应用开发，由于浏览器版本较多，需要将 ECMAScript 6 转译为 ECMAScript 5 的语法运行，其实这跟 CoffeeScript 的使用方式是一样的，因此这种情况下 ECMAScript 6 只能作为语法糖使用，实际运行时不能真正使用到新版本标准的特性；二是 Node 端的应用开发，由于 Node 环境对新版本特性支持较为完善，因此可以使用 ECMAScript 6 的新特性，尤其是对于完善或增强类特性的支持，能够大大提升开发效率，完善功能实现效果。在这种情况下，开发者对 CoffeeScript 的使用慢慢减少，渐渐转向遵循规范的 ECMAScript 6+（泛指 ECMAScript 6 以上版本）标准。

### 3.3.3 TypeScript 概况

我们可以认为，JavaScript 除了主要遵循 ECMAScript 6+ 标准外，还遵循另一种语法规则，即 TypeScript。TypeScript 是微软在 2012 年推出的一种自由开源编程语言，是 JavaScript 的一个

超集。尽管 ECMAScript 新标准和草案不断推出，但 TypeScript 依然存在，同时也获得了不少追捧者。TypeScript 除了部分独有的新特点外，依然主要遵循 ECMAScript 6+标准的规范。

TypeScript 相对于 ECMAScript 6+标准的差异性很小，大部分与 ECMAScript 6+保持一致。为此一部分开发者在接受 TypeScript 概念的欣喜之余也开始反省，TypeScript 提出的差异性特性优势并不明显，在实现 ECMAScript 6+特性支持的基础上增加了少数特殊应用场景下优势的内容。当然，笔者个人觉得 TypeScript 今后的发展仍有待观察。

### 3.3.4 JavaScript 衍生脚本

什么是 JavaScript 的衍生脚本？目前可以理解为基于现有 JavaScript 的实现扩展自己特有语法规则来适应特殊应用场景的一类脚本规范。也可以理解为 JavaScript 的超集，并且通常需要额外支持自身特性的解析器来转译成 JavaScript 解析执行，例如 JSX 或 HyperScript。CoffeeScript 和 TypeScript 在某种意义上也是一种 JavaScript 衍生脚本，但这是之前的版本了，以后可能还会出现其他新的衍生脚本来适应相对应的场景。

例如 JSX 语法的官方声明，使用它的原因是其定义了更简洁且支持对应 DOM 属性的树状结构描述语法。它的标记规则类似于 HTML，但解析不完全一样。HyperScript 则是另一种可以方便创建 Virtual DOM（Virtual DOM，也叫虚拟 DOM，虚拟 DOM 是用来描述 HTML DOM 节点之间属性和对应关系的一类 JavaScript 对象，通常在内存里是一个简单对象，通过特定的规则解析可以与原有 HTML DOM 进行对应的转换，在后面的章节中会展开介绍）的描述性脚本语言，也是 JavaScript 的一种超集。

但是不管怎么样，不同的 JavaScript 衍生脚本均具有一些共性。

- 基于 JavaScript，是 JavaScript 的超集
- 适用于特定的场景
- 具有自己的规范
- 可以按一定的规则解释运行或转译成 JavaScript 运行

或者可以认为，任何一种语言的新版本都是前一个版本的衍生脚本。是否遵循语言的长期发展规范也决定着这个衍生脚本规范能否继续下去。例如 CoffeeScript 的出现衍生了 ECMAScript 5 的标准语法，ECMAScript 6 规范却又借鉴了 CoffeeScript 的部分特性和一些其他的语言优势然后将 CoffeeScript 淘汰。这也说明了开发过程中遵循语言标准的重要性，或许某一

天新的 ECMAScript 标准将直接支持现在衍生脚本创建 Virtual DOM 功能的语法。

前端脚本语言的演进过程主要包括以下几个阶段：ECMAScript 5、CoffeeScript、ECMAScript 6+、TypeScript 和衍生脚本。本节中我们大致了解了它们的特点和应用场景，那么在下一节中，我们将重点来介绍 JavaScript 标准在实际应用开发中的特性应用和表现。

## 3.4 JavaScript标准实践

JavaScript 标准自确定后就和 ECMAScript 有着密切的联系，目前几乎所有的 JavaScript 代码都遵循 ECMAScript 规范。可以简单理解为 JavaScript 是语言，ECMAScript 则是语言习惯。

如果将 JavaScript 比作英语，那么 ECMAScript 标准可以理解为美式英语，TypeScript 则可以理解为英式英语。它们大部分的语法都是相同的，但仍有少部分的差异，都是英语的使用标准。

这一节中，我们将从 ECMAScript 5 开始讲起，重点了解一下 ECMAScript 标准是如何一步步改进完善的。2009 年 12 月，ECMAScript 5.0 版正式发布。Harmony 项目（Harmony 项目其实可以认为是未被发布的 ECMAScript 4.0 版本，由于草案发生分歧，最后被命名 Harmony）一分为二：一些较为可行的语言特性被命名为 JavaScript.next 继续开发，并演变成后来的 ECMAScript 6；另一些不是很成熟的语言特性，则被视为 JavaScript.next.next 版本，在更远的将来再考虑推出。2015 年 6 月 17 日，ECMAScript 6 正式发布，因此 ECMAScript 6 也被称为 ECMAScript 2015，此版本标准在原有的 ECMAScript 特性上进行了较大的修改和完善，现已成为较为通用的 JavaScript 开发标准。2016 年，ECMAScript 7 发布。下面逐个来了解这些与 JavaScript 标准相关的内容。

### 3.4.1 ECMAScript 5

ECMAScript 5 于 2009 年 12 月发布，内容主要包括严格模式、JSON 对象、新增 Object 接口、新增 Array 接口和 Function.prototype.bind。可以认为 ECMAScript 5 规范的推出在原来没有规范的 JavaScript 语法上添加了有限的限制标准。其中最重要的一条可能就是严格模式的提出。

#### 👉 严格模式

ECMAScript 5 严格模式的提出为开发者提供了更加安全规范的编程范围，限制了原有一些不规范的写法，让一些不合理的语法直接报错，从而提高了代码的安全性和规范性。例如，在

严格模式下，未声明的全局变量赋值会抛出 `ReferenceError`，而不是默认去创建一个全局变量；默认支持的糟糕特性都会被禁用，使用 `with`、`eval` 等语句执行会抛出 `SyntaxError`；限制了函数中的 `arguments` 使用，严格模式下函数内部的 `arguments.callee()` 和 `arguments.caller()` 调用会提示 `Uncaught ReferenceError`；删除系统内置对象属性、对象已冻结(`isFrozen`)的属性、对象已密封(`isSealed`)的属性、全局变量或 `var` 定义的变量等都是不允许的，否则会报 `Uncaught SyntaxError` 错误；对禁止扩展的对象添加新属性或对对象只读属性赋值会报 `Uncaught TypeError`；函数参数不能有重名的情况，否则会报 `Uncaught SyntaxError`；函数 `arguments` 严格定义为参数，不再与形参绑定，尽管如此，我们依然不建议使用 `arguments` 变量；`call/apply` 的第一个参数将直接传入不自动封装为对象；对象属性不建议有重名的情况发生，否则一部分浏览器会报 `Uncaught SyntaxError`，所以也不建议出现。具体来看下面的这些例子。

```
'use strict';
myName = 'ouven'; // Uncaught ReferenceError: myName is not defined
person = {        // Uncaught ReferenceError: person is not defined
  name: myName,
  address: 'China',
  job: 'engineer',
  job: 'writer', // 部分浏览器报错: Uncaught SyntaxError
  city: 'Shenzhen',
  website: 'http://www.jixianqianduan.com'
};

with(person){ // Uncaught SyntaxError: Strict mode code may not include a with statement
  console.log(name, address, job, city, website);
  eval('alert(name) ');
};

delete myName; // Uncaught SyntaxError: Delete of an unqualified identifier in strict mode.

function sayHi(arguments){ // Uncaught SyntaxError: Unexpected eval or arguments in strict mode
  console.log(`Hi ${arguments}`);
}
```

需要注意的是，严格模式下错误的提示类型和描述在不同浏览器可能不相同，具体与浏览器的实现有关。

总体来说，严格模式的添加消除了 Javascript 语法的一些不合理、不严谨之处，减少了一些怪异行为，可以在一定程度上提高编译器效率，加快运行速度，为未来新版本的 Javascript 标准化做了铺垫，这是非常有意义的。

## 📌 JSON

一般情况下, 我们使用 JavaScript 语言解析字符串为 JSON 对象或解析 JSON 对象为字符串时可以使用 `JSON.parse()` 和 `JSON.stringify()`, 这些用法相信大家也都了解。问题是, 我们要知道 JSON 对象解析不是伴随着 JavaScript 的出现而产生的。例如在比 IE8 更低版本的浏览器中不能直接使用 JSON 的解析方法。不过现在我们通常可以在浏览器中添加 es5-shim 来增加浏览器对 ECMAScript 5 功能的支持, 让浏览器支持 JSON 对象的解析, 这样我们就可以在后面的代码中放心使用 `JSON.parse()` 和 `JSON.stringify()` 了。

```
<script src="//www.domain.com/es5-shim.js"></script>
```

JSON 对象中有几个容易混淆的方法: `JSON.stringify()`、`JSON.toString()`、`JSON.valueOf()`、`JSON.toLocaleString()`。`JSON.stringify()` 适用于将 JSON 内容转为字符串; `JSON.valueOf()` 用于获取某个对象中的值; `JSON.toString()` 被调用时会调用 `Object` 原型上的 `toString` 方法, 会取得 JSON 对象的值并转为字符串, 如果没有具体的值, 则返回原型数组; `JSON.toLocaleString()` 也是 `Object` 原型上的方法, 经常会返回与 `toString()` 相同内容, 但对于 `Date` 对象, `toLocaleString()` 会返回格式化后的时间字符串。

```
JSON.stringify({name: 'ouven'}); // 输出: '{"name":"ouven"}"
JSON.toString({name: 'ouven'}); // 输出: "[object Object]"
JSON.valueOf({name: 'ouven'}); // 输出: JSON {Symbol(Symbol.toStringTag): "JSON"} 对象
JSON.toLocaleString({na006De: 'ouven'}); // 输出: "[object object]"

let colors = ['red', 'blue', 'green'];
console.log(colors.toString()); // red, blue, green
console.log(colors.valueOf()); // ["red", "blue", "green"]
console.log(colors.toLocaleString()); // red, blue, green

let date = new Date();
console.log(date.toString()); // Thu Sep 08 2016 10:07:50 GMT+0800 (中国标准时间)
console.log(date.valueOf()); // 1473300470759
console.log(date.toLocaleString()); // 2016/9/8 上午 10:07:50
```

## 📌 新增Object方法属性

根据 ECMAScript 5 规范文档, ECMAScript 5 标准添加了较多 `Object` 原型对象上的方法。

ECMAScript 5 标准中新增的 `Object` 方法和属性如下, 见表 3-2。尽管这些新增的 `Object` 属性或方法在我们实际的业务开发中并不一定常用, 但如果要自己抽象一个工具类或者实现 JavaScript 库, 就可能会用到它们, 这里读者们可以灵活选择。

表 3-2 ECMAScript 5 Object 新增方法

方 法	描 述
getPrototypeOf	返回一个对象的原型
getOwnPropertyDescriptor	返回某个对象自有属性的属性描述符
getOwnPropertyNames	返回一个数组，包括对象所有自有属性名称集合（包括不可枚举的属性）
create	创建一个拥有指定原型和若干指定属性的对象
defineProperty	为对象定义一个新属性，或者修改已有的属性，并对属性重新设置 getter 和 setter，这里可以被用作数据绑定的对象劫持用途
defineProperties	在一个对象上添加或修改一个或者多个自有属性，与 defineProperty 类似
seal	锁定对象。阻止修改现有属性的特性，并阻止添加新属性，但是可以修改已有属性的值
freeze	阻止对对象的一切操作和更改，冻结对象将变为只读
preventExtensions	让一个对象变的不可扩展，也就是不能再添加新的属性
isSealed	判断对象是否被锁定
isFrozen	判断对象是否被冻结
isExtensible	判断对象是否可以被扩展
keys	返回一个由给定对象的所有可枚举自身属性的属性名组成的数组

```
let colors = ['red', 'blue', 'green'];

console.log(Object.getPrototypeOf(colors)); // [Symbol(Symbol.unscopables): Object]对象
console.log(Object.getOwnPropertyDescriptor(colors)); // undefined
console.log(Object.getOwnPropertyNames(colors)); // ["0", "1", "2", "length"]

Object.seal(colors); // 锁定对象
Object.freeze(colors); // 冻结对象
Object.preventExtensions(colors); // 阻止扩展

console.log(Object.isSealed(colors)); // true
console.log(Object.isFrozen(colors)); // true
console.log(Object.isExtensible(colors)); // true
console.log(Object.keys(colors)); // ["0", "1", "2"]

colors[2] = 'gray'; // Uncaught TypeError: Cannot assign to read only property '2' of object '[object Array]'
colors[4] = 'gray'; // Uncaught TypeError: Can't add property 4, object is not extensible
delete colors[2]; // Uncaught TypeError: Cannot delete property '2' of [object Array]
```

📌 新增Array方法属性

ECMAScript 5 标准对内置数组对象的原型方法也做了扩展完善，主要添加了下列常用的方法，如表 3-3 所示。

表 3-3 ECMAScript 5 Array 新增方法

方 法	描 述
indexOf	返回根据给定元素找到的第一个索引值，如果不存在则返回-1
lastIndexOf	返回指定元素在数组中的最后一个索引值，如果不存在则返回-1
every	测试数组的所有元素是否都通过了指定函数的测试
some	测试数组中的某些元素是否通过了指定函数的测试
forEach	令数组的每一项都执行指定的函数
map	返回一个由原数组中每个元素调用某个指定方法得到的返回值所组成的新数组，返回每一个处理结果
filter	利用所有通过指定函数处理的元素创建一个新的数组并返回
reduce	接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终缩减为一个值
reduceRight	接受一个函数作为累加器，数组中的每个值（从右到左）开始缩减，最终缩减为一个值

相比 Object 新增的内容，内置对象 Array 新增的这些方法都是很方便很常用的，为对数组进行操作提供了便捷，方便对业务数据进行处理。

```
let colors = ['red', 'blue', 'green', 'green'];

console.log(colors.indexOf('green')); // 2
console.log(colors.lastIndexOf('green')); // 3

console.log(colors.every(function(color){
    return color.length >= 3;
})); // true, 判断是否所有的元素长度均大于等于 3

console.log(colors.some(function(color){
    return color.length > 4;
})); // true, 判断是否有至少一个元素长度大于 4

colors.forEach(function(color){
    if(color === 'green'){
        console.log(color);
    }
}); // green green

console.log(colors.map(function(color){
    if(color === 'green'){
        return color;
    }
})); // 输出数组: [undefined, undefined, "green", "green"]

console.log(colors.filter(function(color){
    if(color === 'green'){
```

```

        return color;
    }
})); // 输出数组: ["green", "green"]

console.log(colors.reduce(function(color, nextColor){
    return color + ',' + nextColor;
})); // 输出字符串: red,blue,green,green

console.log(colors.reduceRight(function(color, nextColor){
    return color + ',' + nextColor;
})); //输出字符串: green,green,blue,red

```

### 👉 Function.prototype.bind

ECMAScript 中新增的函数的 `bind()` 方法比较常用，`bind()` 方法会创建一个新函数，称为绑定函数。当调用这个绑定函数时，绑定函数会以创建它时传入 `bind()` 方法的第一个参数作为 `this`，以传入 `bind()` 方法的第二个以及以后的参数和绑定函数运行时本身的参数按照顺序作为原函数的参数来调用，具体如下。

```
fun.bind(thisArg[, arg1[, arg2[, ...]]]);
```

其实 JavaScript 中重新绑定 `this` 变量的函数方法还有 `call()`、`apply()`，不过 `bind()` 显然与它们有着明显的不同，`bind()` 会返回一个新的函数，并将传入的参数和函数绑定起来，而 `call()` 或 `apply()` 则是使用新的 `this` 去直接调用、执行函数。

```

const fun = function(param) {
    console.log(this+ ':' +param)
};
fun.call(this, 'ouven'); // " [object Window]:ouven"
fun.apply(this, ['ouven']); // " [object Window]:ouven"

let funNew = fun.bind('ouven', 'zhang');
funNew(); // 'ouven:zhang'

```

### 👉 String.prototype.trim、Date.now()、Date().toJSON

```

console.log(' ouven zhang '.trim()); // "ouven zhang"
console.log(Date.now()); // 返回当前时间戳，如 1473303676336
console.log((new Date()).toJSON()); // "2016-11-21T02:59:28.599Z"

```

总体来说，ECMAScript 5 增加的特性相对不多，主要是对之前语法的完善，但大多新增内特性都是很实用的，相信大部分大家都已经熟悉了。接下来我们重点看一下 ECMAScript 6 标准的新特性和一些需要注意的问题。



### 3.4.2 ECMAScript 6

ECMAScript 6 于 2015 年 6 月 17 日正式发布, 也被命名为 ECMAScript 2015。早在草案制定阶段, ECMAScript 6 特性就被部分开发者应用在实际项目中。ECMAScript 6 借鉴了 ECMAScript 5 和其他语言的特性, 并在此基础上进行了补充和增强, 最终形成了一套完整的特性集合, 使得 JavaScript 语言规范更加高效、严谨、完善。例如字符串模板、集合、箭头函数、Promise、for...of 循环等均是借鉴其他语言的优秀特性而增加的功能点, class 类和 import/export 模块规范则可以认为是对原有 ECMAScript 标准缺失特性的补充, 迭代器、生成器、解构赋值、函数参数等可认为是对原有标准特性的增强。这里我们将用简短的篇幅来向大家介绍 ECMAScript 6 新增的核心特性的使用方法和实际开发中需要注意的地方。

#### 📌 块级作用域变量声明关键字let、const

```
let a = 1;
const b = 'hello';
var A = 2;

{
  let c = 'c';
  const d;    // Uncaught SyntaxError: Missing initializer in const declaration
}

console.log(c); // Uncaught ReferenceError: c is not defined
b = 'world';    // Uncaught TypeError: Assignment to constant variable.

console.log(window.a || global.a); // undefined
console.log(window.A || global.A); // 2
```

有几个需要注意的地方: 一是 let 和 const 都只能作为块级作用域变量的声明, 且只能在块级作用域内生效, 块内声明的变量无法在块级外层引用; 二是使用 const 声明的变量必须进行初始化赋值, 而且一旦赋值就不能再进行二次修改赋值; 三是使用 let、const 在全局作用域下声明的变量不会作为属性添加到全局作用域对象里面, 这点和 var 是不同的; 四是通过测试, 使用 let、const 赋值语句的执行速度比使用 var 快约 65% 左右。所以, 我们除了知道使用 let 和 const, 也需要了解 let 和 const 使用场景的区别: 模块内不变的引用和常量, 一般使用 const 定义; 可变的变量或引用使用 let 声明; var 仅用于声明函数整个作用域内需要使用的变量。

#### 📌 字符串模板

字符串模板设计主要来自其他语言和前端模板的设计思想, 即当有字符串内容和变量混合连接时, 可以使用字符串模板进行更高效的代码书写并保持代码的格式和整洁性。如果没有字

字符串模板，我们依然需要像以前一样借助“字符串+操作符”拼接或数组 `join()` 方法来连接多个字符串变量。

```
let name = 'ouven';
let str = `

## 


```

要注意的是，字符串模板不会压缩内部的换行与空格，而是按照原有的格式输出，只将变量内容填充替换掉。实际的项目开发中，如果使用 ECMAScript 6 的转译工具将 ECMAScript 6 的代码处理生成 ECMAScript 5 的代码后运行，格式可能丢失，因为 ECMAScript 5 及之前的版本中字符串是没有字符串模板格式的。

### 👉 解构赋值

个人认为，解构赋值是 ECMAScript 6 的一个亮点功能，它解决了赋值的编码冗余和模块按需导出的问题。解构赋值主要分为数组解构和对对象解构。

```
let [a, b, c] = [11, 22];
let {one, two, three} = {two:2, three:3, one:1};

[a, b] = [b, a]; // 交换 a、b 变量的值
console.log(c); // undefined
```

这里数组解构是严格按照数组下标依次对应顺序赋值的，如果赋值的常量个数不够，则对应下标的变量默认为 `undefined`；如果常量个数超出，则多余的会被舍弃，所以顺序很重要；而对对象解构赋值则是根据对象引用的键名来进行赋值的，可以无视顺序。另外，如果某个变量已经被声明，就不能再参加解构赋值了，JavaScript 执行引擎会把它当作重复声明处理，而 ECMAScript 6 中任何变量的重复声明是绝对不允许的，所以会直接提示 `Uncaught TypeError` 错误。

### 👉 数组新特性

ECMAScript 6 也为数组内置对象添加了较多的新特性，主要包括...复制数组和新增的数组 API。

```
const arr = ['ouven', 'zhang'];
const newArr = [...arr]; // ['ouven', 'zhang']
```

需要注意的是，这里的...进行的数组复制是浅拷贝（关于深拷贝和浅拷贝的区别此处不再赘述），而新增的数组方法则主要包括以下几种，如表 3-4 所示。

表 3-4 ECMAScript 6 数组新增方法

方 法	描 述
Array.from	用于将类数组对象（包括对象[array-like object]和可遍历对象）转化为真正的数组
Array.of	可以将传入的一组参数值转换为数组
Array.prototype.copyWithin	可以在当前数组内部将指定位置的数组项复制到其他位置，然后返回当前数组，使用 copyWithin 方法会修改当前数组
Array.prototype.fill	使用给定值，填充一个数组，会改变原来的数组
Array.prototype.find	用于找出第一个符合条件的数组元素，有点类似于 filter
Array.prototype.findIndex	用来返回某个特定数组元素在数组中的位置

除此之外，ECMAScript 6 中还添加了更多关于数组迭代的方法：entries()、keys()和 values()，均可用来遍历数组。它们都返回一个迭代器对象，也可以用 for...of 循环进行遍历，区别是 keys()是对数组键名进行遍历、values()是对数组键值进行遍历，entries()是对数组中键值对进行遍历，另外 Array.prototype[Symbol.iterator]也可以用来获取遍历数组对象的迭代器。

```
let colors = ['red', 'blue', 'green', 'green'];

console.log(Array.from(colors)); // ["red", "blue", "green", "green"]
console.log(Array.of('red', 'blue', 'green', 'green')); // ["red", "blue", "green", "green"]

console.log(colors.find(function(color) {
  if (color === 'green') {
    return color;
  }
})); // green

console.log(colors.findIndex(function(color) {
  if (color === 'green') {
    return color;
  }
})); // 2

console.log(colors.fill('black')); // 会改变 colors 的值，["black", "black", "black", "black"]

// 恢复原数组后再操作
colors = ['red', 'blue', 'green', 'green'];
console.log(colors.copyWithin(0, 3)); // ["green", "blue", "green", "green"]

console.log(colors.keys()); // 输出 ArrayIterator {}对象
```

```
console.log(colors.entries()); // 输出 ArrayIterator {} 对象
```

## 👉 函数参数

ECMAScript 6 对函数参数进行了新的设计，对原有函数参数设计糟糕的部分进行改善，主要添加了默认参数、不定参数和拓展参数。

```
// 默认参数
function sayHi(name = 'ouven'){
  console.log(`Hello ${name}`);
}
sayHi(); // Hello ouven

// 不定参数
function sayHi(...name){
  // 这里 name 的值为['ouven', 'zhang']
  console.log(name.reduce((a,b)=>`Hello ${a} ${b}`));
}
sayHi('ouven', 'zhang'); // Hello ouven zhang

// 扩展参数
let name = ['ouven','zhang'];

function sayHello(name1, name2){
  console.log(`Hello ${name1} ${name2}`);
}

sayHello(...name); // Hello ouven zhang
```

通过实例大家应该都掌握了这些方法，其中不定参数和扩展参数可以认为恰好是相反的两个模式，不定参数是使用数组来表示多个参数，扩展参数则是将多个参数映射到一个数组。同时也有几个地方需要注意，这里不定参数的...和数组复制的...是有区别的，不定参数可以使用函数的形参来表示所有的参数组成的列表。以前的 `arguments` 变量也有类似的作用，但是 `arguments` 不是真正的数组，除了存放参数的列表外，`arguments` 还有 `length` 属性，严格来说 `arguments` 是一个类数组对象，而不定参数则是一个完全的数组，这也是不定参数相对于 `arguments` 的优势，更加方便我们使用，所以建议使用不定参数来代替 `arguments`。当然，如果在 ECMAScript 6 中定义了'use strict'，`arguments` 的使用也基本被限制了，所以我们应该尽量使用新的函数参数。

## 👉 箭头函数

箭头函数的设计来自于 CoffeeScript 等语言的特性，ECMAScript 6 标准中也添加了这个特性，让短函数的声明更加方便。

```
// 箭头函数
[1, 2, 3].forEach((x) => x * x);

(() => {
  console.log('Hello ouven zhang.');// Hello ouven zhang.
})();
```

需要注意的是，箭头函数没有完整的执行上下文，因为其 `this` 和外层的 `this` 相同，可以理解为它的执行上下文只有变量对象和作用域链，没有 `this` 值。

在 JavaScript 中，代码的执行上下文由变量对象、作用域链和 `this` 值组成。但箭头函数与外层执行上下文共享 `this` 值。如果需要创建具有独立上下文的函数，就不要使用箭头函数。

### 👉 增强对象

在 ECMAScript 6 中，对象的使用变得更加方便。你可以在定义对象时通过属性简写、变量作为属性名或省略对象函数属性的书写等方式来提高编码的效率。下面的书写方式就简洁多了。

```
const name = 'ouven';

const people = {
  // 属性简写
  name,
  // 返回变量或对象作为属性名
  [getKey('family')]: 'zhang',
  // 对象方法属性简写声明
  sayHi(){
    console.log(`Hello ${this.name} ${this.family}`);
  }
}

people.sayHi(); // Hello ouven zhang

function getKey(key){
  return key;
}
```

这里没有太多需要注意的地方，但为了使代码便于维护和理解，还是建议尽量不将变量或对象作为对象的属性名，这样不容易阅读。

### 👉 类

JavaScript 没有类这一点一直是 ECMAScript 设计里比较糟糕的地方，这也导致了在 ECMAScript 6 以前定义类的方法以及类的继承方式多种多样。就类的继承方式来说，基本思路就有原型链继承、构造函数继承、实例继承和拷贝继承几种，但每种方法多多少少都有自己的

缺陷，因为它们都不是真正的类继承，例如会出现子类不一定是父类的实例、子类和父类共享一个实例等糟糕的情况。ECMAScript 6 添加了 `class` 关键字，一切便都有章可循了。

```
class Amina1{
  constructor() {
    // ...
  }
}

class People extends Amina1{
  constructor(contents = {}) {
    super();
    this.name = contents.name;
    this.family = contents.family;
  }
  sayHi() {
    console.log(`Hello ${this.name} ${this.family}`);
  }
}

let boy = new People({
  name: 'ouven',
  family: 'zhang'
});

boy.sayHi(); // Hello ouven zhang
```

当然有了 `class`，就有 `extends`，对于开发者来说，使用 `class` 很大的好处是实现一个类的代码模块只能在一个地方定义，想想以前我们可以在代码中的任意位置去扩展基类的 `prototype` 属性，从某种意义上来说，`class` 类声明解决了这个之前设计糟糕的地方，对代码的规范性和严谨性都提出了更好的限制方案。

## 📁 模块module

ECMAScript 6 引入了模块引用规范，这也是之前语言标准上没有的。这样现有的 JavaScript 模块化规范又多了一种选择：`import/export`。

```
// 简单的模块导入导出示例
import { sayHi } from './people';
export default sayHi;
```

需要注意的是，使用 `default` 导出时要尽量将其他模块需要使用的部分属性导出，不要导出整个对象，因为这样会导出一些不需要的东西，如果使用模块按需内容导出，部分 ECMAScript 6 的打包工具可以使用静态树分析的方法来自动移除程序运行时不需要的代码，例如支持 Tree-shaking 实现方式的思路，将原有代码中未被导出或未使用到的代码部分作为无效代码移除

掉，这样在开发后期模块打包优化的时候就很有用了。

JavaScript 之前的模块化规范比较多，包括 AMD、CMD、CommonJs，现在又多了 ECMAScript 6 的 import/export。所以为了统一，选择模块开发规范时应尽量遵循语言标准的特性。

### 👉 循环与迭代器Iterator

到了 ECMAScript 6 阶段，JavaScript 实现循环的方式就比较多了，除了 do...while、for 循环，还可以使用 for...in 来遍历对象（注意不要用 for...in 来遍历数组，因为遍历出来的键不是数字，而且在部分浏览器中会产生乱序）。如果遍历数组的话，则可以使用 for...of、map、forEach，需要注意的是，使用 map、forEach 这类方式的一个好处是将循环遍历的数组元素指定到一个具体的处理函数中进行处理，这样的函数一般是个纯函数，但缺陷是用它时不能直接跳出整个循环，只能跳出当前单步循环。所以数组循环遍历最佳方式是 for...of，此外 for...of 也可以用来遍历 Map、Set、WeakMap、WeakSet 等集合。Iterator 迭代器的加入则让遍历数组、对象和集合的方式更加灵活可控，Iterator 可以控制每次单步循环触发的时机，不用一次遍历所有的循环。

纯函数一般指返回值完全由传入的参数来决定的函数，即对于同一个参数输入，在任何情况下的函数返回结果都是相同且唯一的。例如数组的 slice() 可以认为是一个纯函数，而 random() 则不是纯函数。

以数组遍历为例（其他对象遍历的方法与此类似），我们先看一下直接迭代的方式和用 Iterator 迭代器实现的方式有什么区别。

```
// for...of 遍历实现
const numbers = [1, 2, 3, 4, 5];

for(let number of numbers){
  console.log(number);
}

// 迭代器遍历数组
const numbers = [1, 2, 3, 4, 5];

let iterator = numbers[Symbol.iterator]();
let result = iterator.next();
console.log(result.value); //1

//...doSomething1
result = iterator.next();
```

```
console.log(result.value); //2

//...doSomething2
result = iterator.next();
console.log(result.value); //3

//...doSomething3
result = iterator.next();
console.log(result.value); //4

//...doSomething4
result = iterator.next();
console.log(result.value); //5
```

相信大家很快便看懂 **Iterator** 和 **for...of** 的区别了。**Iterator** 可以在循环开始后任意的地方进行数组的单步循环，当循环迭代中每次单步循环操作都不一样时，使用 **Iterator** 就很有用了。如果使用 **for...of** 则需要不断判断执行的次数来执行不同的单步循环。需要注意的是，这里也可以使用 **Iterator** 来循环一次性遍历数组所有的元素，每次 **Iterator** 调用 **next()** 都会返回一个对象 `{done: false, value: item}`，**done** 属性是 **boolean** 值，表示循环遍历是否完成，**value** 则是每一步 **next()** 调用获取到的值。

如图 3-6 所示，为了方便理解，我们可以把 **Iterator** 理解成为数组或对象上的一个根据偏移来访问内存内容的游标对象，每次调用 **next()**，遍历游标会向后移动一个地址。另外，**Symbol.iterator** 的方法属性是在 **Object** 原型上的，所以可以用来遍历一切可循环遍历的对象。

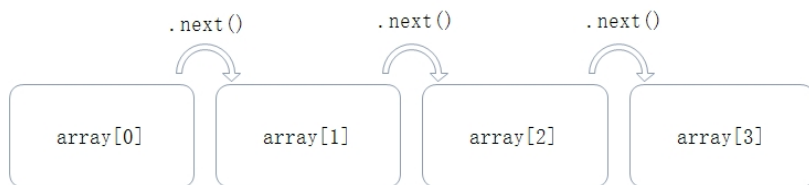


图 3-6 Iterator 调用运行示意图

## 👉 生成器Generator

如果对 **Iterator** 理解较深的话，那么你会发现生成器 **Generator** 和 **Iterator** 的流程是有点类似的。但是，**Generator** 不是针对对象上内容的遍历控制，而是针对函数内代码块的执行控制，如果将一个特殊函数的代码使用 **yield** 关键字来分割成多个不同的代码段，那么每次 **Generator** 调用 **next()** 都只会执行 **yield** 关键字之间的一段代码。**Generator** 可以认为是一个可中断执行的特殊函数，声明方法是在函数名后面加上 **\*** 来与普通函数区分。上面数组遍历的例子，使用



Generator 书写如下。

```
const generator = function* (){
  const numbers = [1, 2, 3, 4, 5];
  for(let number of numbers){
    yield console.log(number);
  }
}

let result = generator();

result.next(); // 1
//...doSomething
result.next(); // 2
//...doSomething
result.next(); // 3
//...doSomething
result.next(); // 4
//...doSomething
result.next(); // 5
```

代码很浅显易懂，不过需要注意的是，Generator 遇到 `yield` 关键字会暂停往后执行，但并不表示后面的程序就不执行了。如果 `console.log(number)` 是一个耗时的工作，那么程序只在 Generator 里面暂停，外面的程序仍会继续执行，举例如下。

```
const generator = function* (){
  const numbers = [1, 2, 3, 4, 5];
  for(let number of numbers){
    yield setTimeout(function(){
      console.log(number);
    }, 3000);
  }
}

let result = generator();
let done = result.next();
while(!done.done){
  done = result.next();
}
console.log('finish');
```

执行上述代码会先输出 `finish`，然后才打印数字，结果如下。

```
finish
1
2
3
4
5
```

这样不就可以用来控制多个异步操作了吗！尤其对于 Node 服务端处理浏览器的请求，这样来处理异步显得更为优雅。

总结一下实现数组或对象循环遍历的方法：for、while(do...while)、forEach、map、reduce、for...of、for...in、Iterator、Generator。除了这些可用的方式，大家也要尽量去了解实现这些循环的差异性和对应的应用场景。到目前为止，推荐使用 for...of 或 for...in 来处理大多数的循环遍历场景。

### 📌 集合类型 Map + Set + WeakMap + WeakSet

也许很多人会疑惑，既然数组和对象可以存储任何类型的值，为什么还需要 Map 和 Set 呢？考虑几个问题：一是对应的键名一般只能是字符串，而不能是另一个对象；二是对象没有直接获取属性个数等这些方便操作的方法；三是我们对于对象的任何操作都需要进入对象的内部数据中完成，例如查找、删除某个值必须循环遍历对象内部的所有键值对来完成。总之我们使用简单对象的方式仍然显得很低效，没有一个高效的方法集来管理对象数据。因此 ECMAScript 6 增加了 Map、Set、WeakMap、WeakSet，试图弥补这些不足。这样我们就可以使用它们提供的 has、add、delete、clear 等方法来管理和操作数据集合，而不用具体进入到对象内部去操作了，这种情况下 Map 和 Set 就类似一个可用于存储数据的黑盒，我们只管向里面高效存取数据，而不用知道它里面的结构是怎样的。我们甚至可以这样理解：集合类型是对对象的增强类型，是一类使数据管理操作更加高效的对象类型。

如果一定要对应来看，Set 可以认为是增强的数组类型，Map 则可以认为是增强的对象类型，WeakSet 和 WeakMap 则对应着 Set 和 Map 的优化类型，所以某种程度上，为了让程序开发更加方便，我们有必要引入集合这类更为高效的类型。WeakSet 和 WeakMap 在生成时有更加严格的限制：WeakSet 只存储对象类型的元素，不能遍历，没有 size 属性；WeakMap 只接受基本类型的值作为键名，没有 keys、values、entries 等遍历方法，也没有 size 属性。

```
let ws = new WeakSet();
ws.add({ data: 42 });

let wm = new WeakMap();
wm.set('key', { data: 42 });
// ...这里不再一一列举它们的具体用法
```

需要注意的是，Map 和 Set 都为内部的每个键或值保持了强引用，也就是说，如果一个存储的属性元素被移除了，回收机制可能无法回收它占用的内存，容易造成内存泄露，所以

我们使用时要尽可能先删除引用的相关内容。相比之下，使用 WeakSet 和 WeakMap 则不会出现上述情况，因为它们并不使用强引用。再总结一下 JavaScript 可能出现内存泄露的常见场景：闭包函数、全局变量、对象属性循环引用、DOM 节点删除时未解绑事件、Map 和 Set 的属性直接删除。希望大家同时也要明白上述五种场景的具体情况是怎么样的。

## 👉 Promise、Symbol、Proxy增强类型

### Promise

接触过 Promise 的都知道，Promise 可以用来避免异步操作函数里的多层嵌套回调（callback hell）问题，因为解决多层异步场景最直接的方法是回调嵌套，将后一个操作放在前一个操作的异步回调里，但如果操作层数多了，就会很难管理。

Promise 的出现很好地解决了这一问题，Promise 代表一个异步操作的执行返回状态，这个执行返回状态在 Promise 对象创建时是未知的，它允许为异步操作的成功或失败指定处理方法。目前 Promise 实现的方式较多，一般将这些 Promise 实现的规范分为 Promise/A 规范和 Promise/A+ 规范，通常我们认为 Promise/A+ 规范是在 Promise/A 规范的基础上进行修正和增强形成的，更具有规范性。

我们怎样区分 Promise/A+ 规范和 Promise/A 规范呢？

- 符合 Promise/A+ 规范的 promise 实现一般以 then 方法为交互核心。Promise/A+ 维护组织也会去解决 Promise 中新发现的问题并改进完善规范，因此 Promises/A+ 规范相对来说更加稳定。
- Promise/A+ 规范要求 onFulfilled 或 onRejected 返回 promise 后的处理过程必须是作为函数来调用，而且调用过程必须是异步的。
- Promise/A+ 规范严格定义了 then 方法链式调用时 onFulfilled 或 onRejected 的调用顺序。

判断是否为 Promise/A+ 规范主要看 Promise 方法是否含有 new Promise(function(resolve, reject){})、then、resolve、all 等方法。另外 ECMAScript 6 中 Promise 的实现严格遵循了 Promise/A+ 规范，而 jQuery 中的 Deferred 就不是 Promise/A+ 规范。

```
// Deferred 精简后主要逻辑的源码
Deferred: function( func ) {
    let tuples = [
        // action, add listener, listener list, final state
        [ "resolve", "done", jQuery.Callbacks("once memory"), "resolved" ],
```

```

    [ "reject", "fail", jQuery.Callbacks("once memory"), "rejected" ],
    [ "notify", "progress", jQuery.Callbacks("memory") ]
  ],
  state = "pending",
  promise = {
    state: function() {},
    always: function() {},
    then: function( /* 成功调用, 失败调用, 处理中状态 */ ) {},
    promise: function( obj ) {}
  },
  deferred = {};
  jQuery.each(tuples, function( i, tuple ){
    deferred[tuple[0] + "With"] = list.fireWith;
  });
  promise.promise(deferred);
  return deferred;
}

```

从上面 jQuery 的代码中可以看出, jQuery 中 Deferred 实现其实是一个工厂函数, 执行 \$.Deferred 之后返回一个带有 state、always、then、promise 方法的对象, 它的状态描述也作为自己的属性保存在对象中。而 ECMAScript 6 中 Promise 的实例化后内容一般只用来描述状态。

```

// ECMAScript 6 的 promise
{[[PromiseStatus]]: "resolved", [[PromiseValue]]: "Fulfilled"}

```

通常 Promise 的状态有三种: **Fulfilled** 状态表示 Promise 执行成功; **Rejected** 状态表示 Promise 执行失败; **Pending** 状态表示 Promise 正在执行中。

```

let status = 1;
let promise = new Promise(function(resolve, reject){
  if(status === 1){
    resolve('Fulfilled');
  }else{
    reject('Rejected')
  }
});
promise.then(function(msg){
  console.log('success1:' + msg);
  return msg;
},function(msg){
  console.log('fail1:' + msg);
  return msg;
}).then(function(msg){
  console.log('success2:' + msg);
},function(msg){
  console.log('fail2:' + msg);
});

```

我们来看一个具体的例子，`promise` 的 `then` 方法接受两个处理函数，当 `status` 为 1 时执行 `Fulfilled` 成功调用，否则执行 `Rejected` 失败调用。然后将返回的状态返回给第二个 `then` 方法处理，并将状态打印。输出结果分别如下。

```
success1: Fulfilled
success2: Fulfilled
```

`then` 方法可以将传入参数的返回值一直传递下去，如果是异步的场景，就可以用这种方式来解决多层回调嵌套的问题了。

```
// 典型的异步 promise 例子，希望它依次异步输出 A、B、C、D
let promise = new Promise(function(resolve) {
  setTimeout(function() {
    console.log('A');
    resolve();
  }, 3000); // 延时 3 秒打印 A
});
// 使用 then 来链式处理流程
promise.then(function() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      console.log('B')
      resolve();
    }, 2000); // 延时 2 秒打印 B
  });
}).then(function() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      console.log('C')
      resolve();
    }, 1000); // 延时 1 秒打印 C
  });
}).then(function() {
  return new Promise(function(resolve, reject) {
    console.log('D'); // 不延时打印 D
  });
});
```

这是一个经典的处理异步嵌套的例子，需要按照顺序依次异步输出 A、B、C、D，这种情况可以通过在不同的 `then` 处理方法中返回一个新的 `Promise` 来解决。返回新的 `Promise` 里面具有 `resolve()` 和 `reject()` 方法，只有当它的 `resolve` 或 `reject` 被调用时，`Promise` 方法才会继续执行，进入下一个 `then` 方法中操作。如果我们设置在异步函数完成的最后调用 `resolve()` 就可以有效控制 `Promise` 进入下一个 `then` 方法执行了。理解了这个示例，就基本了解了 `Promise` 处理多层异步的回调机制。

需要注意的是，新的版本浏览器放弃了 ECMAScript 6 中 `Promise.defer()` 的使用方式，因为它并不是 Promise/A+ 标准方法。尽管使用 `defer` 来处理这个问题会更简单。

## Symbol

Symbol 是 ECMAScript 6 新增加的基本数据类型，一般用作属性键值，并且能避免对象属性键的命名冲突。

```
let object = {};  
let name = Symbol();  
let family = Symbol();  
  
object[name] = 'ouven';  
object[family] = 'zhang';  
  
/*  
 * 这里 object 的值是这样的，显示属性的键名是 Symbol，这就是 Symbol 的作用，你仍可以使用 object[name]  
 * 去读取对象的属性  
 * {  
 *   Symbol(): 'ouven',  
 *   Symbol(): 'zhang',  
 * }  
 */  
console.log(object);  
console.log(typeof name); //symbol
```

这里因为对象的键是 Symbol 变量，而 Symbol 变量是不能被重复声明的，这种情况下对象属性定义时属性键就不会被重复定义了。

在 ECMAScript 6 中，JavaScript 的数据类型就有七种了：null、undefined、boolean、string、number、symbol、object。

## Proxy

ECMAScript 6 新增的另一个特性是 Proxy。Proxy 可以用来拦截某个对象的属性访问方法，然后重载对象的“.”运算符。这似乎和我们之前讲到的某种方法很相似，即 `defineProperty` 中的 `getter` 和 `setter`，举例如下。

```
let object = new Proxy({}, {  
  get: function (target, key, receiver) {  
    console.log(`getting ${key}`);  
    return Reflect.get(target, key, receiver);  
  },  
  set: function (target, key, value, receiver) {
```

```

        console.log(`setting ${key}`);
        return Reflect.set(target, key, value, receiver);
    }
});

```

// 看看之前的一个方法

```

let object = {};
let value;

```

```

Object.defineProperty(object, 'value', {
    // 重写 get 方法
    get: function() {
        console.log('getting value');
        return value;
    },
    // 重写 set 方法
    set: function(newValue) {
        value = newValue;
        console.log('setting: ' + newValue);
    },
    enumerable: true,
    configurable: true
});

```

// 赋值或定义值都会输出

// getting value

// setting value

```
object['value'] = 3;
```

这两段代码的作用类似，都是劫持并重定义对象的 `getter` 和 `setter` 方法进行自定义返回（这种实现方式通常也叫对象劫持），一部分前端 MVVM（Model-View-ViewModel，后面的章节我们会详细讲解）框架的数据变更检测就是通过此手段实现的，现在通过实现 `Proxy` 也可实现和 `defineProperty` 类似的功能，当然，这只是 ECMAScript 6 中 `Proxy` 的一种比较典型的方法。

## 👉 统一码

ECMAScript 6 字符串支持新的 Unicode 文本形式，同时也增加了新的正则表达式修饰符 `u` 来处理统一码。尽管如此，在实际的开发中，这样处理仍会降低程序可读性和处理速度，所以目前不建议使用。

```
'字符串'.match(/./ug).length === 3 //true
```

### 👉 进制数支持

ECMAScript 6 增加了对二进制 (b) 和八进制 (o) 数字面量的支持。可能这个在实际开发中很少遇到, 所以大家只要先了解就可以了。

```
0b111110111 === 503 // true
0o767 === 503 // true
```

### 👉 Reflect对象和tail calls尾调用

Reflect 可以理解为原有对象上的一个引用代理, 它用于对原有对象进行赋值或取值操作, 但不会触发对象属性的 getter 或 setter 调用, 而直接使用=对对象进行赋值或取值操作会自动触发 getter 或 setter 方法, 关于 Reflect 的用法可以参考上面 Proxy 的例子。

tail calls 尾调用保证了函数尾部调用时调用栈有一定的长度限制, 这使得递归函数即使在没有限制输入时也能保证安全性而避免发生错误。

```
function factorial(n, start = 1) {
  if (n <= 1) {
    return start;
  }
  return factorial(n - 1, n * start);
}
// 默认情况下会发生栈溢出, 但是在 ECMAScript 6 中是可以安全执行的
factorial(100000);
```

关于 ECMAScript 6 的核心特性主要就这么多, 本节用尽可能少的篇幅系统地介绍了 ECMAScript 6 的所有新特性和常用特性在使用中容易犯的错误及需要注意的地方, 如果想要了解更详细的关于 ECMAScript 6 设计与实现的内容, 可以去查阅更完整的书籍资料。

参考资料: <http://kangax.github.io/compat-table/>。

## 3.4.3 ECMAScript 7+

2016 年, ECMAScript 7 (或称 ECMAScript 2016) 正式发布。整体来说, 在 ECMAScript 6 这个历史性版本逐渐稳定之后, 后期版本添加的主要内容已经不是太多了, 主要包含幂指数操作符与 Array.prototype.includes, 下面我们具体来看看 ECMAScript 7 的内容。

### 👉 幂指数操作符

```
x**y 产生的结果等同于 Math.pow(x, y)
console.log(2**3); //8
```



增加了新的操作符来进行幂指数运算，但实际上，这种情况在前端开发中并不是很常用。

### 👉 Array.prototype.includes

这个数组方法主要用来判断数组中是否包含某个元素。

```
let colors = ['red', 'blue', 'green', 'green'];
console.log(colors.includes('green')); // true
```

除了以上特性，仍有部分新内容将在 ECMAScript 7 后推出，这些也是讨论比较多的，我们将选取部分较为典型的来讲解。

### 👉 异步函数async/await

在 ECMAScript 6 发布时，部分特性的实现方案并没有按期发布，大家便预想这些特性会在下个版本中出现，其中被关注比较多的就是异步函数了，那么我们一起来看看 ECMAScript 7 中的异步函数是怎样的。对比一下 ECMAScript 6 中 Generator 的例子。

```
const asyncFunction = async function () {
  const numbers = [1, 2, 3, 4, 5];
  for (let number of numbers) {
    await sleep(3000);
    console.log(number);
  }
}

let result = asyncFunction();
console.log('finish');
```

输出结果为：

```
finish
1
2
3
4
5
```

我们惊奇地发现，异步函数的写法与 Generator 相比其实是非常类似的，区别在于 async 函数将 Generator 函数的星号\*替换成 async，将 yield 替换成 await，并且少了 next() 的调用控制。实际上也确实如此，我们可以认为 async/await 是对 Generator 的一种封装简化，专门用于处理 Generator 中异步的场景。毕竟 Generator 可以使用 next() 来更加灵活地控制整个程序流程的执行，处理异步只是一种使用情况。

### 📌 SIMD.JS -- SIMD APIs + Polyfill

SIMD (single instruction, multiple data) 代表的是单指令多数据流, 涉及并行计算范畴的语法指令, 可以看出未来 JavaScript 有可能会在并行计算领域内使用。但该特性目前在前端开发中没有具体适合的使用场景, 未来在服务器端则可能成为一个常用的增强特性。

我们再来总结一下 JavaScript 中实现异步的方法: `setTimeout`、事件监听、观察者模式、`$.Deferred`、`promise`、`generator`、`async/await`、第三方 `async` 库等。

参考资料: <https://tc39.github.io/ecma262/>。

## 3.4.4 TypeScript

关于 ECMAScript 6 和 ECMAScript 7 的内容我们先了解这么多, 下面来了解另一种 JavaScript “方言”——TypeScript。TypeScript 是 2012 年微软发布的一种开源语言, 和与之结合的开源编辑器 VS code (Visual Studio Code) 一起推出供开发者使用。到今天, TypeScript 也已经发生了比较大的变化, 就语言特性来说, TypeScript 基本和 ECMAScript 6 的语法保持一致, 可以认为是 ECMAScript 6 的超集, 基本包含了 ECMAScript 6 和 ECMAScript 6 中部分未实现的内容, 例如 `async/await`, 但仍有一些少数的差异性特征。

### 📌 强类型支持

TypeScript 的数据类型是强类型的, 声明时需要对类型进行定义。这种规范在某种程度上避免了由于不同类型之间的变量赋值修改所导致的问题, 但就目前的开发方式和应用场景来说, 该特性仍没有很好的应用优势, 反而增加了使用的复杂度。

```
let name:string = 'ouvenzhang';
let fullName:String = new String(name);
```

### 📌 Decorator装饰器特性

Decorator 可以用来注解 `class`、`property`、`method` 和 `parameter`, 也是一种面向对象编程语言设计模式的借鉴, 目前新版的 Angular 2 框架也引入了 TypeScript 语法和 Decorator 使用来描述代码复用性定义。

```
class Amina1{
  constructor() {
    // ...
  }
}
```

```
class People extends Amina1{
  constructor(contents = {}) {
    super();
    this.name = contents.name;
    this.family = contents.family;
  }
  @sayHi(this)
}

function sayHi(self) {
  console.log(`Hello ${self.name} ${self.family}`);
}

let boy = new People({
  name: 'ouven',
  family: 'zhang'
});

boy.sayHi(); // Hello ouven zhang
```

相对于 ECMAScript 6 而言, TypeScript 的定位比较尴尬, 目前开发者对它的评价也褒贬不一, 毕竟凭借一两个亮点功能并不能表现出与 ECMAScript 6+规范的差异化优势, 对于它未来的发展性, 仍有待进一步验证。

## 3.5 前端表现层基础

### 3.5.1 CSS发展概述

CSS (Cascading Style Sheets) 是随着前端表现分离的提出而产生的, 因为最早网页内容的样式都是通过<center>、<strike>标签或 fontColor 等属性内容来体现的, 而 CSS 的设计则提出使用样式描述语言来表达页面内容, 而不是用 HTML 的标签来表达。关于 CSS 基础的部分相对不多, 我们将从 CSS2 开始了解。

继 CSS1 后, W3C 在 1998 年发布了 CSS2 规范, CSS2 的出现主要是为了解决早期网页开发过程中排版时表现分离的问题, 后来随着页面表现的内容越来越复杂, 浏览器平台厂商继续推动 W3C 组织对 CSS 规范进行更多的改进和完善, 添加了例如 border-radius、text-shadow、transform、animation 等更灵活的表现层特性, 逐渐形成了一套全新的 W3C 标准, 即 CSS3。CSS3 可以认为是在 CSS2 规范的基础上进行补充和增强形成的, 让 CSS 体系更能适应现代浏览器的需要, 拥有更强的表现能力, 尤其对于移动端浏览器, 目前的移动端浏览器主要以 Google Chrome 和 Apple Safari 浏览器的 webkit 内核为主, 由于浏览器内核版本相对较高, 可以更容易地支持

CSS3 特性来完成更多、更复杂的事情。当然目前 CSS4 的草案也在制定中，CSS4 中更强大的选择器、伪类和伪元素特性已经被曝光出来，但具体发布时间仍不确定。

要了解 CSS，主要还是要了解 CSS 的属性和内容，那么我们就从几个方面来快速把握 CSS 相关知识的整体脉络。

### 3.5.2 CSS选择器与属性

#### 👉 CSS选择器

简单回顾一下 CSS 有哪几类选择器：id 选择器、类选择器、元素选择器、组合选择器、伪类、伪元素等。一般认为 CSS 中选择器属性优先级顺序为 !important > 内联样式（权重 1000）> id 选择器（权重 100）> 类选择器（权重 10）> 元素选择器（权重 1），多种组合情况按照权重相加的原则来计算，!important 优先级最高。

```
<style>
#element{
    display: inline-block;
}

.ui-btn{
    display: inline-block;
}

.ui-btn:hover{
    color: red;
}

div{
    display: block;
    width: 200px;
    height: 100px;
}

button{
    display: block !important;
}
</style>
<div style="color: red;"></div>
```

另外值得注意的是，元素定义的 CSS 伪类和伪元素是不同的。简单地说，伪元素会在 HTML 中添加 before 或 after 这类内容，而像 :visited、:hover、:first-child、:nth-child、:enable、

:checked 这些伪类则不会，一般用于表示元素在用户不同操作下的状态或选择指定某些元素的描述，所以读者们要注意区分伪类和伪元素。

### 👉 CSS属性

CSS 的属性和值直接决定着元素在页面上的渲染表现形式，关于 CSS 常用属性我们可以按照表 3-5 所示的方式进行归类。

表 3-5 CSS 属性分类

属性类型	属 性 名
布局类属性	position 类、弹性布局 flex、浮动 float、对齐 align
几何类属性	盒模型相关(margin、padding、width、height、border)、box-shadow、渐变 gradient、background 类、transform 类
文本类属性	font 类、line-height、color 类、text 类(text-decoration、text-indent、textoverflow)、white-space、user-select、text-shadow 等
动画类属性	以 css3 为主的 transition、animation 等
查询类	Media query 和 IE Hack 等

尽管我们觉得 CSS 的属性很多，但是分类以后发现，基本就以上几类，此外均是一些相对不常用的属性。这里的属性使用比较简单，此处就不一一展开了。

对于开发者来说，在 CSS 编码开发中，复杂的或许并不是学习这些属性对应的值有哪些，而是在实际项目中处理 CSS 兼容性问题。因为浏览器的版本、平台的多样性导致浏览器对 CSS 属性支持的差异性极大，兼容性问题总是很难完美解决，遇到这些情况就只能冷静下来一步步分析，经验的积累尤为重要。

### 3.5.3 简单的应用举例

CSS 在页面中的实现非常灵活，通常实现一个效果的方式会有很多，我们需要根据具体的场景选择最合适的方式来实现。这里举两个较典型的简单例子。

弹性布局。例如我们要实现一个等宽的三列布局，并根据父元素的宽度自动改变，实现的思路有很多。

```
<ul>
  <li>菜单 1</li>
```

```
<li>菜单 2</li>
<li>菜单 3</li>
</ul>

<!-- 常见的方式 -->
<style>
*{
    margin: 0;
    padding: 0;
}
ul{
    width: 100%;
}
ul li{
    float: left;
    width: 33.333%;
    list-style-type: none;
}
</style>

<!-- flex 弹性布局的方式 -->
<style>
*{
    margin: 0;
    padding: 0;
}
ul{
    display: -webkit-box;
    display: -webkit-flex;
    display: -ms-flexbox;
    display: flex;
}

ul li{
    -webkit-box-flex: 1;
    -moz-box-flex: 1;
    -webkit-flex: 1;
    -ms-flex: 1;
    flex: 1;
    list-style-type: none;
}
</style>
```

除此之外，还有其他的实现方式，可以灵活考虑。再如页面中省略号的显示，我们要在页面中实现一行或两行文字超出时显示省略号的效果，就可以这样来写。

```
// 实现一行文字省略号
p{
```

```
width: 200px;
height: 36px;
overflow: hidden;
text-overflow: ellipsis;
white-space: nowrap;
}

// 实现两行文字省略号
p{
width: 200px;
height: 36px;
overflow: hidden;
text-overflow: ellipsis;
display: -webkit-box;
display: -webkit-flex;
display: -ms-flexbox;
display: flex;
-webkit-line-clamp: 2;
line-clamp: 2;
-webkit-box-orient: vertical;
box-orient: vertical;
}
```

第二种实现方式需要考虑兼容性的问题，推荐用于移动端，另外也可以通过 JavaScript 截取的方式来达到类似的效果。再如实现元素内容居中可以选择 `margin: 0 auto` 设置居中、`text-align: center` 设置居中、`display: table-cell; vertical-align: middle` 设置垂直居中、绝对定位等多种方法。

这里列举了几个比较简单的例子，前端开发中需要处理 CSS 的典型场景很多，还有清除浮动、自适应三列布局等一些典型的实现案例大家都可以去了解，此处不一一展开了。

关于 CSS 的标准规范基础这里讲的内容相对较少，并不是说 CSS 就比较简单，这些是我们进行开发的基础，而且了解这些仍然不够。实际工程实践中，我们还需要结合 CSS 的预处理技术和组件化 UI 框架的设计理念来高效地实现具体功能的表现模块，这些内容将会在下一节中具体介绍。

## 3.6 前端界面技术

上一节中我们简单了解了前端表现层的基础内容，这一节中我们来结合实际的开发技术进行项目实践。前端界面技术主要指目前网页表现层上的开发实现技术。下面从以下几个方面来了解现代前端的界面技术：前端表现层 CSS 样式统一化、预处理技术、表现层动画实现，最后

我们一起来简单了解 CSS4 的发展情况。

### 3.6.1 CSS样式统一化

目前访问 Web 网站应用时,用户使用的浏览器版本较多,由于浏览器间内核实现的差异性,不同浏览器可能对同一元素标签样式的默认设置是不同的,如果不对 CSS 样式进行统一化处理,可能会出现同一个网页在不同浏览器下打开时显示不同或样式不一致的问题。要处理这一问题目前主要有三种实现思路: reset、normalize 和 neat。

#### 👉 reset

先来看看使用 reset 的方式进行样式统一化的思路:将不同浏览器中标签元素的默认样式全部清除,消除不同浏览器下默认样式的差异性。典型的 reset 默认样式的代码如下。

```
body, h1, h2, h3, h4, h5, h6, hr, p, blockquote, dl, dt, dd, ul, ol, li, pre, form, fieldset,
legend, button, input, textarea, th, td{
    margin:0;
    padding:0;
}
```

这种方式可以将不同浏览器上大多数标签的内外边距清除,需要注意的是,这个例子中的规则不能消除标签所有的差异性,而这里只是针对消除内外边距差异性来举例子,其他样式的差异性处理方法类似。消除默认样式后重新定义元素样式时,常常需要针对具体的元素标签重写样式来覆盖 reset 中的默认规则,所以这种情况下我们常常需要去重写样式来对元素添加各自的样式规则。

例如元素标签<li>默认会有列表样式,而我们通常并不需要,那么就可以在 reset 中统一设置 list-style-type: none 来处理;另外<li>与<li>之间可能有回车空格会导致元素之间在浏览器上有空白间隙,这样的问题也可以通过在 reset 中统一设置 float 等方法来解决。

#### 👉 normalize

相对于 reset 方式存在的问题,normalize 的思路稍有区别,normalize 的做法是在整站样式基本确定的情况下对标签元素统一使用同一个默认样式规则,例如整个站点规定元素之间的最小常用的内外边距都是 5px,则代码如下。

```
body, h1, h2, h3, h4, h5, h6, hr, p, blockquote, dl, dt, dd, ul, ol, li, pre, form, fieldset,
legend, button, input, textarea, th, td{
    margin:5px;
    padding:5px;
}
```



```
}
```

这种方式建议在网站基本样式或规范确定的情况下使用。相对于 `reset`，这种实现方式避免了较多的样式重写情况，例如使用 `reset` 上面的这种情况就要在每个出现的元素里面定义 `margin:5px; padding:5px;`，但一般 `normalize` 只适用于网站前端基本设计规范确定且统一的情况下，否则 `normalize` 将失去意义而且会导致页面表现层样式不易维护。

### neat

`neat` 可以认为是对上面两种实现的综合，因为我们通常不能保证网站界面上的所有元素的内外边距都是确定的，又不想将所有样式都清除后再进行覆盖重写。例如我们只想对文本相关元素设置内外边距 `5px`，而其他的元素则清除不同浏览器的默认样式差异，那么这样定义就显得比较整洁。

```
body, dl, dt, dd, ul, ol, li, form, fieldset, button, input, textarea{
    margin:0;
    padding:0;
}

h1, h2, h3, h4, h5, h6, hr, p, blockquote, dt, pre, legend,textarea, th, td{
    margin:5px;
    padding:5px;
}
```

总结来说，前端统一化 CSS 样式主要有 `reset`，`normalize`，`neat` 三种实现思路：`reset` 是清除浏览器的默认样式并保持在所有浏览器中一致；`normalize` 是使用同一种默认样式并在所有浏览器中保持样式一致；`neat` 则可以认为是前两种的结合，具体需要根据网站的设计特点来确定，但仍需要保证默认样式在所有浏览器中是一致的。三种方法各有自己的使用场景，我们可以结合具体的团队项目开发模式来选择，通常情况下由于网页界面的设计是不确定的，所以目前使用 `reset` 的开发场景比较多。

一个典型的 CSS `reset` 的实现代码如下。

```
body, h1, h2, h3, h4, h5, h6, hr, p, blockquote, dl, dt, dd, ul, ol, li, pre, form, fieldset,
legend, button, input, textarea, th, td, article, aside, details, figcaption, figure, footer,
header, hgroup, menu, nav, section{
    margin:0;
    padding:0;
}

html, body{
    width:100%;
    height: 100%;
}
```

```
h1 {
  font-size: 18px;
}
h2 {
  font-size: 16px;
}
h3 {
  font-size: 14px;
}
h4, h5, h6 {
  font-size: 100%;
}
a {
  text-decoration: none;
}
a:hover {
  text-decoration: underline;
}
fieldset, img {
  border: none;
}
table {
  border-collapse: collapse;
  border-spacing: 0;
}
:focus{
  outline: 0;
  -webkit-tap-highlight-color: transparent;
}
```

### 3.6.2 CSS预处理

网页前端表现层通常是直接使用 CSS 来实现的，目前 CSS 开发也已经进入了预处理时代。CSS 预处理工具有很多，例如 SASS、LESS、Stylus、postCSS 等。尽管使用的工具不同，而且各有特点和优势，但所有预处理工具的最终目的是一致的：通过编写更高效、易管理的类 CSS 脚本并将它们自动生成浏览器解释执行的 CSS 代码，现实高效开发和便捷管理。所以 CSS 预处理技术具有几个优势：提高开发效率，例如 SCSS 的嵌套、父级选择符、Mixin、Extend，或者 postCSS 的 autoprefixer 等都是为了减少书写重复性代码、提高编写效率而设计的；便于管理则，预处理器使用的类 CSS 脚本可以按模块编写开发进行管理，而且幸运的是大部分预处理工具都有相应的模块引用机制并能借助构建工具自动完成编译打包。这样我们就可以专注于业务模块层中 CSS 脚本的开发了。

这里我们还是以 SASS（或许有人认为有点落后，但工具本身并没有那么重要）为例来看

看这些预编译工具有能实现什么功能。其他一些预处理工具是类似的，或许还带有一些更加便捷的功能。

SASS 的预处理实现主要包含模块引用、嵌套、父级选择符、嵌套属性、注释、变量、数据类型、运算、圆括号、函数、篡写、变量默认值、规则指令、控制指令、mixin、继承 extend 等这些属性。具体如下。

```
// 可以方便地引入模块
@import "reset.scss";

// 变量赋值与计算能力，可以统一管理样式重出现的重复变量或默认样式
$width: 5px + 10px;
$height: 5rem;
$name: box;
$attr: margin;
$content: 'empty text' !default;
$maxWidth: 375px;

// 处理函数
@function getWidth($n){
  @return $n*3rem - 1rem;
}

// mixin代码块。这里与处理函数的区别在于，mixin的内容会被全部填充到引入的元素代码里面，而function
// 函数只做过程处理并输出
@mixin mod{
  width: $width;
  height: ($height + 3rem)/2;
  $font-size: 12px;
  $line-height: 20px;
  // 可以使用#{ }包住变量进行计算
  font: #{ $font-size }/#{ $line-height };
}

.ui-mod-a{
  // 设立使用@include引用的mixin内容将被填充进来
  @include mod;
  &:hover{
    cursor: pointer;
  }
  &:after{
    content: $content;
  }
  .p.#{ $name }{
    #{ $attr }-left: 4px;
  }
}
```

```
@media screen and(max-width: $maxWidth){
  #{$attr}-left: 8px;
}

// 继承的使用, 这里.ui-mod-b 继承了.ui-mod-a 的所有属性, 并且覆写了 width 属性
.ui-mod-b{
  @extend .ui-mod-a;
  @if null {width: $maxWidth;}
  width: getWidth(3);
}
```

这里用简单的代码基本演示了 SASS 所有的主要功能, 整体上理解也很简单。如果需要进一步了解可以查看手册, 相信大家就能很快明白怎么运用 SASS。我们再看一个 postCSS 的案例了解一下什么是 autoprefixer (自动填充)。

```
.autoprefixer {
  display: flex;
  transform: tranlateZ(0);
}
```

在上面的例子中, postCSS 在处理一些需要兼容性处理的样式时会自动添加兼容性修饰, 不需要我们在编写代码规则时重复编码, 这样就很高效率便捷。通过 postCSS, 上面的代码预处理后最终输出结果如下。

```
.autoprefixer {
  display: -webkit-box;
  display: -webkit-flex;
  display: -ms-flexbox;
  display: flex;
  -webkit-transform: translateZ(0);
  -ms-transform: translateZ(0);
  -o-transform: translateZ(0);
  transform: translateZ(0);
}
```

整体上来看, CSS 预处理器的语法特性和模式的设计很像编程语言的设计思路, 开发完成后通过语法处理器来解析编译成标准规范的 CSS, 这也和之前讲到的 JavaScript 脚本开发的思路很像, 使用简洁高效的衍生语法 (例如 CoffeeScript 语法或浏览器端的 ECMAScript 6+语法) 编译生成安全、规范的 JavaScript 来运行。一个高效的预处理语法工具一般具有以下特性。

- 变量声明和计算。方便一次赋值和随处使用, 并能进行简单运算, 提高开发管理效率。
- 语法表达式。例如 if-else 条件语句、for 循环等简单语法的设计能让页面 CSS 规则的生成更加灵活。

- 函数处理。方便多次计算的地方能统一复用，例如函数处理和 Mixin 等特性。
- 属性的继承。元素类属性的继承在开发样式相似但略微不同的多个模块的过程中非常有用，可以减少大量重复代码。
- 兼容性补全。类似 autoprefixer 这种功能，让开发者不用过多关注不同浏览器的兼容问题，处理多个浏览器兼容性的代码能在预处理阶段自动生成补全，让一些问题的处理方式对开发者透明。

利用这些功能特性，开发类 CSS 脚本相比于原始的 CSS 就高效很多了。尤其是在网站基础结构样式开发时，可以将不同基础功能的实现使用不同的文件来管理实现，不同的组件部分也可以通过不同的文件进行模块化管理。

### 3.6.3 表现层动画实现

除了样式统一化处理和预处理技术，前端界面另一个很重要的内容就是动画，使用符合场景的动画不仅可以优化网站页面中的交互细节，提高用户体验，还可以让页面更具有吸引力，给网站带来更多访问量。通常前端中，实现动画的方案主要有 6 种：JavaScript 直接实现动画、可伸缩矢量图形（Scalable Vector Graphics, SVG）动画、CSS3 transition、CSS3 animation、Canvas 动画、requestAnimationFrame。这样看可能比较抽象，因为大家可能都听说过，但是并不清楚具体细节。下面我们就以一个具体的场景应用为例，来看看这些动画的实现方案。

如图 3-7，我们需要在页面上实现一个方块元素从左向右移动的效果，移动一段距离后停止移动或重新开始移动，上述 6 种方法都可以实现。我们先来看看使用 JavaScript 直接实现动画的方式。

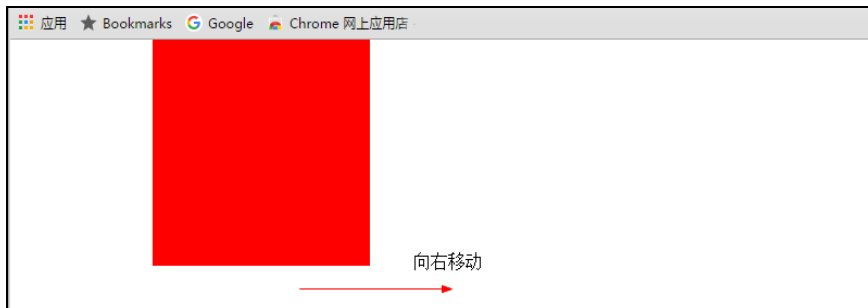


图 3-7 动画效果示例

## 👉 JavaScript直接实现动画

JavaScript 直接实现动画的方式在前端早期使用较多，其主要思想是通过 JavaScript 的 `setInterval` 方法或 `setTimeout` 方法的回调函数来持续调用改变某个元素的 CSS 样式以达到元素样式持续变化的结果，下面是实现一个元素从左向右移动的示例代码。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>动画移动</title>
  <style>
    *{
      margin: 0;
      padding: 0;
    }
    div{
      width: 200px;
      height: 200px;
      background-color: red;
    }
  </style>
</head>
<body>
  <div id="box"></div>
  <script>
    // 获取页面移动元素
    let element = document.getElementById('box');
    let left = 0;

    // 设置定时循环调用改变元素样式的 marginLeft 属性，当到达浏览器右边缘时停止移动
    let timer = setInterval(function(){
      if(left < window.innerWidth - 200){
        element.style.marginLeft = left + 'px';
        left ++;
      }else{
        clearInterval(timer);
      }
    }, 16);
  </script>
</body>
</html>
```

通过代码中的注释，我们很清楚地理解了动画实现的原理和过程。JavaScript 直接实现动画也就是不断执行 `setInterval` 的回调改变元素的 `marginLeft` 样式属性达动画的效果，例如 jQuery 的 `animate()` 方法就属于这种实现方式。不过要注意的是，通过 JavaScript 实现动画通常会导

致页面频繁性重排重绘，很消耗性能，如果是稍微复杂的动画，在性能较差的浏览器上就会明显感觉到卡顿，所以我们尽量避免使用它。

在上面的例子中，有心的读者会发现，我们设置 `setInterval` 的时间间隔是 16ms，为什么呢？一般认为人眼能辨识的流畅动画为每秒 60 帧，这里 16ms 比 1000ms/60 帧略小一点，所以这种情况下可以认为动画是流畅的。在很多移动端动画性能优化时，一般使用 16ms 来进行节流处理连续触发的浏览器事件，例如对 `touchmove`、`scroll` 事件进行节流等。我们通过这种方式来减少持续性事件的触发频率，可以大大提升动画的流畅性。

## 👉 SVG动画

SVG 又称可伸缩矢量图形，原生支持一些动画效果，通过组合可以生成较复杂的动画，而且不需要使用 JavaScript 参与控制。SVG 动画由 SVG 元素内部的元素属性控制，通常通过 `<set>`、`<animate>`、`<animateColor>`、`<animateTransform>`、`<animateMotion>` 这几个元素来实现。`<set>`可以用于控制动画延时，例如一段时间后设置 SVG 中元素的位置，就可以使用 `<set>`在动画中设置延时；`<animate>`可以对属性的连续改变进行控制，例如实现左右移动动画效果等；`<animateColor>`表示颜色的变化，不过现在用 `<animate>`就可以控制了，所以用的基本不多；`<animateTransform>`可以控制如缩放、旋转等几何变化；`<animateMotion>`则用于控制 SVG 内元素的移动路径。我们来看一个元素移动的例子。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>动画移动</title>
  <style>
    *{
      margin: 0;
      padding: 0;
    }
  </style>
</head>
<body>
  <svg id="box" width="800" height="400" version="1.1" xmlns="http://www.w3.org/2000/svg">
    <rect width="100" height="100" style="fill:rgb(255,0,0);">
      <set attributeName="x" attributeType="XML" to="100" begin="4s" />
      <animate attributeName="x" attributeType="XML" begin="0s" dur="4s" from="0"
to="300" />
      <animate attributeName="y" attributeType="XML" begin="0s" dur="4s" from="0"
to="0" />
      <animateTransform attributeName="transform" begin="0s" dur="4s" type="scale"
```

```
from="1" to="2" repeatCount="1"/>
    <animateMotion path="M10,80 q100,120 120,20 q140,-50 160,0" begin="0s" dur="4s"
repeatCount="1"/>
  </rect>
</svg>
</body>
</html>
```

需要注意的是，SVG 内部元素的动画只能在元素内进行，超出<svg>标签元素，就可以认为是超出了动画边界。通过理解上面的代码可以看出，在网页中<svg>元素内部定义了一个边长 100 像素的正方形，并且在 4 秒时间延时后开始向右移动，经过 4 秒时间向右移动 300 像素，同时正方形在移动过程中会放大 1 倍。相对于 JavaScript 直接控制动画的方式，使用 SVG 的一个很大优势是含有较丰富的动画功能，原生可以绘制各种图形、滤镜和动画，绘制的动画为矢量图，而且实现动画的原生元素依然是可被 JavaScript 调用的。然而另一方面，我们要明白，元素较多且复杂的动画使用 SVG 渲染会比较慢，而且 SVG 格式的动画绘制方式必须让内容嵌入到 HTML 中使用。以前这种动画实现的场景相对比较多，但随着 CSS3 的出现，这种动画实现方式相对使用得越来越少了。

## 📌 CSS3 transition

CSS3 出现后，增加了两种 CSS3 实现动画的方式：transition 和 animation，我们先来看看高效强大的 CSS3 过渡动画 transition。为什么称为过渡动画呢？其实 transition 并不能实现独立的动画，只能在某个标签元素样式或状态改变时进行平滑的动画效果过渡，而不是马上改变。依然以元素移动为例来看下面的代码。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>动画移动</title>
  <style>
    *{
      margin: 0;
      padding: 0;
    }
    div{
      width: 200px;
      height: 200px;
      margin-left: 0;
      background-color: red;
      transition: all 3s ease-in-out 0s;
      -webkit-transition: all 3s ease-in-out 0s;
    }
  </style>
</head>
<body>
  <div></div>
</body>
</html>
```



```

        .right{
            margin-left: 400px;
            background-color: blue;
        }

    </style>
</head>
<body>
    <div id="box"></div>
    <script>
        let timer = setTimeout(function() {
            let element = document.getElementById('box');
            // 改变元素的样式
            element.setAttribute('class', 'right');
        }, 500);
    </script>
</body>
</html>

```

这里我们一般通过改变元素的起始状态，让元素的属性自动进行平滑过渡产生动画，当然也可以设置元素的任意属性进行过渡变化。它应用于处理元素属性改变时的过渡动画，而不能应用于处理元素独立动画的情况，否则就需要不断改变元素的属性值来持续触发动画过程了。

在移动端开发中，直接使用 `transition` 动画会让页面变慢甚至变卡顿，所以我们通常通过添加 `transform: translate3D(0, 0, 0)` 或 `transform: translateZ(0)` 来开启移动端动画的 GPU 加速，让动画过程更加流畅。

### 👉 CSS3 animation

CSS3 animation 的动画则可以认为是真正意义上页面内容的 CSS3 动画，通过对关键帧和循环次数的控制，页面标签元素会根据设定好的样式改变进行平滑过渡，而且关键帧状态的控制一般是通过百分比来控制的，这样我们就可以在这个过程中实现很多动画的动作了。定义动画的 `keyframes` 中 `from` 值和 `0%` 的意义是相同的，表示动画的开始关键帧。`to` 和 `100%` 的意义相同，表示动画的结束关键帧。通过 CSS3 animation，上述元素移动效果就可以这样来实现了。

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>动画移动</title>
    <style>
        * {
            margin: 0;
            padding: 0;

```

```
    }

    div {
        width: 200px;
        height: 200px;
        margin-left: 0;
        background-color: red;
        animation: move 4s infinite;
        -webkit-animation: move 4s infinite;
    }
    @keyframes move {
        from {
            margin-left: 0;
        }
        50% {
            margin-left: 400px;
        }
        to {
            margin-left: 0;
        }
    }
</style>
</head>
<body>
    <div id="box"></div>
</body>
</html>
```

相信很多人也很了解了，CSS3 实现动画的最大优势是脱离 JavaScript 的控制，而且能用到硬件加速，可以用来实现较复杂的动画效果。

## 👉 Canvas动画

<canvas>作为 HTML5 的新增元素，也可以借助 Web API 实现页面动画。Canvas 动画的实现思路和 SVG 的思路有点类似，都是借助元素标签来达到页面动画的效果，都需要借助对应的一套 API 来实现，不过 SVG 的 API 可以认为主要是通过 SVG 元素内部的配置规则来实现的，而 Canvas 则是通过 JavaScript API 来实现的。需要注意的是，和 SVG 动画一样，Canvas 动画的进行只能在<canvas>元素内部，超出<canvas>元素边界将不被显示，我们再来看一下前面示例的 Canvas 的实现。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>动画移动</title>
    <style>
```

```

    *{
        margin: 0;
        padding: 0;
    }
</style>
</head>
<body>
    <canvas id="canvas" width="700" height="550">
        浏览器不支持 canvas
    </canvas>
    <script>
        let canvas = document.getElementById("canvas");
        let ctx = canvas.getContext("2d");
        let left = 0;
        let timer = setInterval(function() {
            // 不断清空画布
            ctx.clearRect(0, 0, 700, 550);
            ctx.beginPath();
            // 将颜色块填充为红色
            ctx.fillStyle = "#f00";
            // 持续在新的位置上绘制举行
            ctx.fillRect(left, 0, 100, 100);
            ctx.stroke();
            if (left > 700) {
                clearInterval(timer);
            }
            left += 1;
        }, 16);
    </script>
</body>
</html>

```

元素 DOM 对象通过调用 `getContext()` 可以获取元素的绘制对象，然后通过 `clearRect` 不断清空画布并在新的位置上使用 `fillStyle` 绘制新矩形内容来实现页面动画效果。使用 Canvas 的主要优势是可以应对页面中多个动画元素渲染较慢的情况，完全通过 JavaScript 来渲染控制动画的执行，这就避免了 DOM 性能较慢的问题，可用于实现较复杂的动画。

### 👉 requestAnimationFrame

`requestAnimationFrame` 是前端表现层实现动画的另一种 API 实现，它的原理和 `setTimeout` 及 `setInterval` 类似，都是通过 JavaScript 持续循环的方法调用来触发动画动作的，但是 `requestAnimationFrame` 是浏览器针对动画专门优化而形成的 API，在实现动画方面性能比 `setTimeout` 及 `setInterval` 要好，可以将动画每一步的操作方法传入到 `requestAnimationFrame` 中，在每一次执行完后进行异步回调来连续触发动画效果。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>动画移动</title>
  <style>
    * {
      margin: 0;
      padding: 0;
    }
    div {
      width: 200px;
      height: 200px;
      background-color: red;
    }
  </style>
</head>
<body>
  <div id="box"></div>
  <script>
    // 获取 requestAnimationFrame API 对象
    window.requestAnimationFrame = window.requestAnimationFrame ||
    window.mozRequestAnimationFrame || window.webkitRequestAnimationFrame ||
    window.msRequestAnimationFrame;

    let element = document.getElementById("box");
    let left = 0;
    // 自动执行持续性回调
    requestAnimationFrame(step);

    // 持续改变元素位置
    function step() {
      if (left < window.innerWidth - 200) {
        left += 1;
        element.style.marginLeft = left + "px";
        requestAnimationFrame(step);
      }
    }
  </script>
</body>
</html>
```

可以看出，和 `setInterval` 方法类似，`requestAnimationFrame` 只是将回调的方法传入到自身的参数中处理执行，而不是通过 `setInterval` 调用，其他的实现过程则基本一样。大家有兴趣也可以去比较一下这两种方式的性能消耗情况，`requestAnimationFrame` 是比 `setInterval` 性能消耗低些的。

总的来看,实现动画的方式主要包括这些,这里通过一个简单的案例,向大家介绍了六种不同动画实现方案的基本原理和具体实现。虽然这是个简单的案例,但是复杂动画的实现,也是通过多个简单动画的组合实现的。考虑到兼容性的问题,在项目实践中,一般我们在桌面浏览器端仍然推荐使用 JavaScript 直接实现动画的方式或 SVG 动画的实现方式,移动端则可以考虑使用 CSS3 transition、CSS3 animation、canvas 或 requestAnimationFrame。

了解了动画实现技术,我们接下来简单了解一下前端表现层新版本 CSS4 的情况。

### 3.6.4 CSS4 与展望

目前 CSS 的成熟标准版本是 CSS3,而且在移动端使用较多。CSS4 的规范仍在制定中,W3C 也在较早的时间公布了一些正在制定中的 CSS4 规范,例如 `$e > f`、链接地址伪类 `:any-link` 和 `:local-link`、语言相关伪类 `dir`、新的组分选择器。这些特性我们且先不去关注,因为目前还没看出太多亮点,而且实用性也不是特别强,相比现有的预处理器的语法逊色很多。由于兼容性问题,CSS4 发布后也会处于与 ECMAScript 6 类似的处境(ECMAScript 6 至少还有 Node.js 支持),需要在前端转译后执行,既然都需要转译,那便和现在某个预处理器的语法规则没差别了,要完全兼容恐怕更是遥遥无期。一种可能的最终解决方案是和 ECMAScript 6 一样借鉴现有一些预处理器的优点,整合形成新的规范语法,然后通过预处理器转译为最终的 CSS。这样一个好处是,不用去纠结使用哪个预处理工具,全部以 CSS4 规范为准即可,但这只是一种可能性。

简而言之,CSS4 的处境将会比较尴尬,目前最新的浏览器仍没有支持 CSS4 特性的计划,发布后不能兼容仍需要转译,就目前来看,CSS4 新添加的特性优势并不明显且实用性不强,而且不如现有的预处理语法。所以只能看它后面的发展情况了。

## 3.7 响应式网站开发技术

前面讲解了前端界面技术的 CSS 样式统一化、预处理和动画实现技术,本节我们一起来看看一下现代前端技术中另一个重要的内容——响应式页面实现技术。

### 3.7.1 响应式页面实现概述

通常认为,响应式设计是指根据不同设备浏览器尺寸或分辨率来展示不同页面结构层、行为层、表现层内容的设计方式。谈到响应式设计网站,目前比较主流的实现方法有两种:一是

通过前端或后端判断 `userAgent` 来跳转不同的页面完成不同设备浏览器的适配,也就是维护两个不同的站点来根据用户设备进行对应的跳转;二是使用 `media query` 媒体查询等手段,让页面根据不同设备浏览器自动改变页面的布局和显示,但不做跳转。

先看第一种方案,图 3-8 为目前一种典型响应式站点访问跳转的流程,这里以服务端通过判断请求的 `userAgent` 信息为例。要注意的是,在浏览器端判断 `UserAgent` 执行跳转也是可以的,但是需要页面脚本下载完后再次执行判断跳转逻辑,比服务端执行跳转的方式要慢。用户使用桌面浏览器和移动端浏览器分别可以访问到对应的站点,但是如果使用桌面浏览器直接访问移动站点域名下 Web 站点页面,Web 站点会根据 `userAgent` 信息判断进行 302 跳转到对应的桌面 Web 服务器页面路径下。使用移动设备直接访问桌面端服务器站点的流程与此类似。

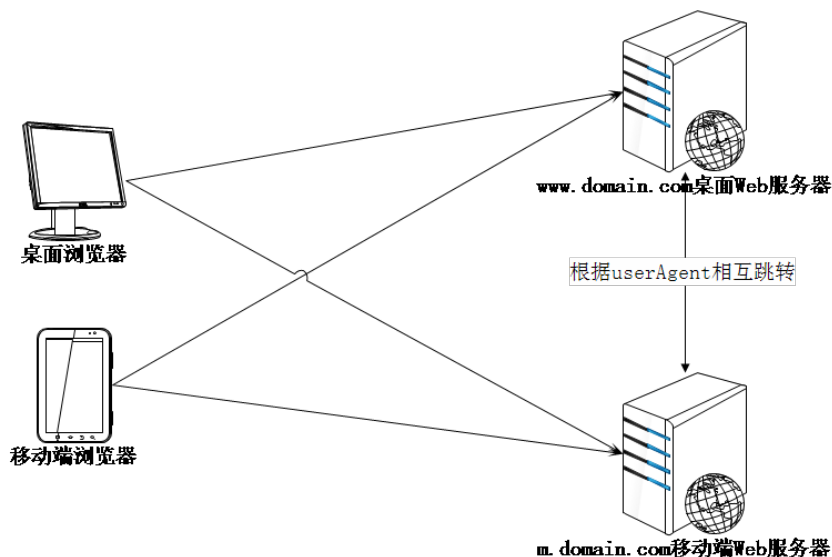


图 3-8 典型响应式站点实现

这样便可以根据不同的设备加载相应的网页资源了,根据这种思路针对移动端的浏览器也可以请求加载更加优化后的执行脚本或更小的静态资源了。根据 `userAgent` 进行跳转会有一定的网络消耗,但是这种模式下必须这样,该实现方案适用于功能复杂并对性能要求较高的站点应用,我们再来具体看一下这种情况存在的一些问题。

- 需要开发并维护至少两个站点跳转来适配不同用户的设备浏览器。例如使用 `www.domain.com` 和 `m.domain.com` 来分别指向桌面浏览器端和移动端浏览器访问的不同 Web 站点服务,然后根据 `userAgent` 来做对应跳转。

- 选择使用哪个站点内容由设备的 `userAgent` 信息来判断,无法根据屏幕尺寸或分辨率来决定。
- 多了一次跳转。无论是在前端执行跳转还是后台执行跳转,这部分逻辑不能少,否则用户体验相会变差。

通常根据浏览器 `userAgent` 来实现跳转的方式也很简单,一般可以在页面头部插入一段 JavaScript 代码来判断或在服务端通过统一的中间件执行跳转。通常,根据 `userAgent` 信息执行页面跳转的代码如下。

```
// 前端页面判断跳转
if (navigator.userAgent.match(/iPhone|iPod|Android|iPad/i)) {
    let hash = window.location.hash,
        params = window.location.search;
    location.href = '//m.domain.com/path/page.html' + params + (hash ? (params ? '&' : '?')
+ hash.substr(1) : '');
}

// Web 服务器端判断跳转
if (this.header['user-Agent'].match(/iPhone|iPod|Android|iPad/i)) {
    let hash = this.hash,
        params = this.querystring;
    this.redirect('//m.domain.com/path/page.html ' + params + (hash ? (params ? '&' : '?')
+ hash.substr(1) : ''));
}
```

再来看第二种方案。桌面浏览器和移动端浏览器使用同一个站点域名来加载内容,只需要开发维护一个站点就可以了,然后根据 `media query` 来实现不同屏幕下的布局显示,适用于访问量较小、性能要求不高的应用场景,例如使用 **Bootstrap** 这类响应式自动布局框架实现的网站。当然,这种方式也存在一些明显的问题。

- 移动端浏览器加载了与桌面端浏览器相同的资源,例如图片、脚本资源等,导致移动端加载到冗余或体积较大的资源。对于网络和计算资源相对较少的移动端来说,这显然不是一个很好的处理方案。虽然可以经过一些优化来减少一部分移动端加载的内容,但是能做的非常有限,不能完全解决这个问题。
- 桌面端浏览器和移动端浏览器访问站点需要展示的内容可能不完全相同,这种响应式的方式只实现了内容布局显示的适应,但是要做更多差异性的功能比较难。
- 桌面端浏览器和移动端浏览器页面功能本身具有差异性,使用同一套处理方式,会有更多的兼容性问题。

响应式页面设计一直是一个很难完美解决的问题，因为多多少少都存在这些问题。带着这些问题，我们再来看看响应式页面设计具体应该怎样做。先来总结一下上面这两种方案的所有问题描述。

- 能否使用同一个站点域名避免跳转的问题
- 能否保证移动端加载的资源内容最优
- 如何做移动端和桌面端浏览器的差异化功能
- 如何根据更多的信息进行更加灵活的判断，而不仅仅是 `userAgent`

经过综合性方案分析，可以得出结论：合理的开发方式和网站访问架构设计是可以解决上述四个问题的。下面我们来看看在响应式的三层结构上具体能做什么处理。

### 3.7.2 结构层响应式

结构层响应式设计可以理解成 **HTML** 内容的自适应渲染实现方式，即根据不同的设备浏览器渲染不同的页面内容结构，而不是直接进行页面跳转。这里页面中结构层渲染的方式可能不同，包括前端渲染数据和后端渲染数据，这样主要就有两种不同的设计思路了：一是页面内容是在前端渲染，二是页面内容在后端渲染（也就是直出层，我们后面会具体讲到这个）。

#### 👉 结构层数据内容响应式

目前仍有较多的网站使用的是前后端分离、前端数据渲染的方式实现的，尤其是移动端页面。这种情况下如果要做到结构层响应式就要考虑到，首先要保证移动端加载的内容资源最小，因此会以移动端优化资源为主，保证移动端页面的首屏内容优先加载，然后通过异步的方式来实现桌面端或移动端剩余内容的加载。

这样似乎就没有问题了，其实这样在最先加载的页面 **HTML** 文件内容中，桌面端浏览器加载的内容中不可避免会有少量的移动端冗余内容，而且值得注意的是，一般下载的 **HTML** 文件内容在桌面端和移动端会有差异化内容存在，这是不可避免的，所以我们只能尽可能减少差异化内容来保证冗余资源减到最小。但即使不能避免，由于桌面端的计算资源和网络资源相对比较宽裕，这些也是可以接受的，这毕竟比移动端浏览器加载桌面端的冗余内容好多了。

具体来看，我们根据不同平台浏览器的情况加载不同的异步静态 **JavaScript**，然后异步渲染不同的模块内容，生成不同的表现层结构就可以如下来实现。



```
// isMobile 是根据 userAgent、屏幕尺寸或屏幕分辨率判断是否为移动端设备的结果
let isMobile = navigator.userAgent.match(/iPhone|iPod|Android|iPad/i);
if(isMobile){
    require.async(['zepto', './mobileMain'], function($, main){
        main.init();
    });
}else{
    require.async(['jQuery', './main'], function($, main){
        main.init();
    });
}

// ./main.js 文件内容
module.exports = {
    init: function(){
        $('#article').html(PC:这是桌面端浏览器内容');
    }
}

// ./mobileMain.js 文件内容
module.exports = {
    init: function(){
        $('#article').html('mobile:这是移动端浏览器内容');
    }
}
```

如图 3-9 所示，桌面端浏览器和移动端浏览器直接加载到的 HTML 结构是相同的，由于页面的数据内容主要在前端渲染，那么就可以使用异步的方式加载桌面端或移动端不同的 JavaScript 资源列表来渲染不同的前端数据内容了。为了保证我们使用移动端打开的页面加载到相对最优的页面资源内容，我们也可以使用异步的方式来加载 CSS 文件，这样就可以做到根据移动端页面和桌面端页面加载到不同的资源内容了。

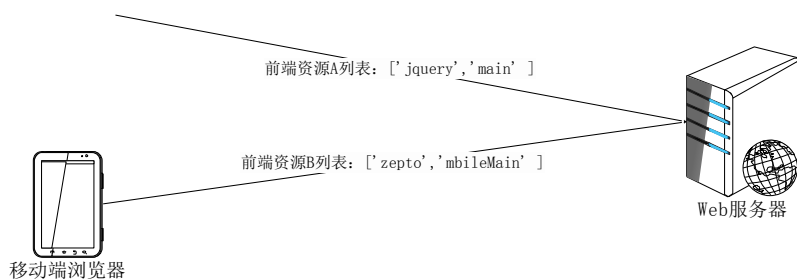


图 3-9 前端内容响应式渲染方式

使用这种方式尽管可以让桌面端和移动端复用一个页面并做到页面的差异化，但是由于使

用了同一个 HTML 结构模板为基础进行渲染和操作，因此页面的功能实现仍然有部分耦合的地方。

### 👉 后端数据渲染响应式

除了前端数据渲染的方式，目前还有一部分网站的内容生成使用了后端渲染的实现方式。这种情况的处理方式其实可以做到更优化，只要尽可能将桌面端和移动的业务层模板分开维护就可以了。在模板选择判断时仍是可以通过 `userAgent` 甚至 `URL` 参数来进行的。

```
// isMobile 是根据 userAgent、URL 参数判断是否为移动端设备的结果
let isMobile = this.headers['user-Agent'].match(/iPhone|iPod|Android|iPad/i);
if (isMobile) {
  res.body = yield render('pages/mobile/main', {
    data: pageData
  });
} else {
  res.body = yield render('pages/pc/main', {
    data: pageData,
    moreData: moreData
  });
}

// pages/mobile/main 模板内容
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>移动端内容</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0,maximum-scale=1.0,user-scalable=no">
  <link rel="stylesheet" href="./mobile/path/main.css">
</head>
<body>
  <section>{{ data || safe }}</section>
  <script src="./mobile/main.js"></script>
</body>
</html>

// pages/pc/main 模板内容
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>桌面端内容</title>
  <link rel="stylesheet" href="./pc/path/main.css">
</head>
```

```

<body>
  <div>{{ data || safe }}</div>
  <div>{{ moreData || safe }}</div>
  <script src="./pc/main.js"></script>
</body>
</html>

```

如图 3-10 所示,直出层以 Node 为例,在生成页面内容时可以判断用户的 `userAgent` 来渲染不同的 HTML 输出模板。这里不同的模板内容可以完全不一样,并可以独立维护,这样就能根据同一个地址直出不同的内容了,例如在桌面端或移动端浏览器中打开 Google 搜索首页时,同一个地址展示的内容却不一样,便是根据这种思路来实现的。当然这里用的模板最好是根据组件化开发分开打包生成的两个不同模板,否则后期维护成本就比较高了。

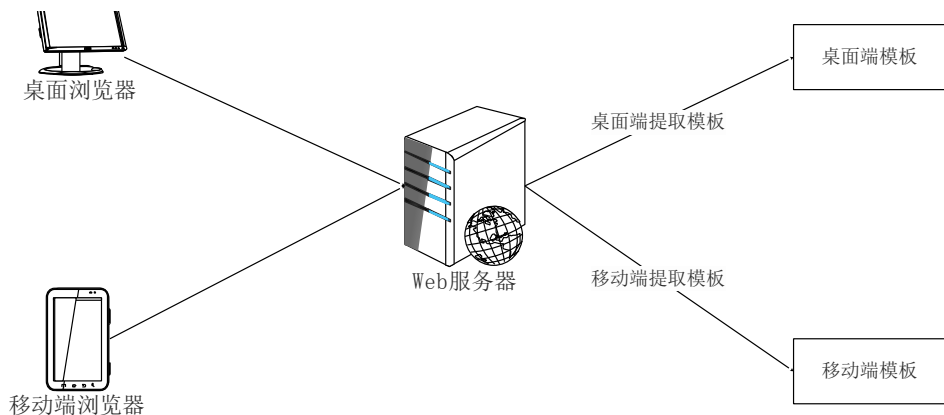


图 3-10 后端内容响应式渲染方式

这种情况下我们就可以完全将桌面端页面和移动端页面结构层分开管理了,不仅可以复用共同的基础组件,还可以差异化开发不同的业务组件,JavaScript 资源和 CSS 资源也是完全分开加载的,实现两个端加载内容的相互独立,就解决了上面描述的所有问题。当然,越接近完美的实现需要付出的代价也往往越大,这种实现方式通常不可避免地要将桌面端和移动端结合起来同步开发。大家也可以根据自己团队的技术架构和具体情况来选择尝试这种开发模式,不仅如此,这种实现思路也很适合大型应用站点的实践。

如图 3-11 所示,使用在直出层响应式渲染输出不同页面内容的模式也很容易结合业务接入层进行大型应用的开发。使用组件化思路,前端工程师可以专注于前端模块的开发,构建后生成直出层的不同的数据模板,直出层则用于调用业务层服务模块来获取处理的数据,然后根据 `userAgent` 判断不同的浏览器设备调用不同的模板渲染出不同的页面结构。

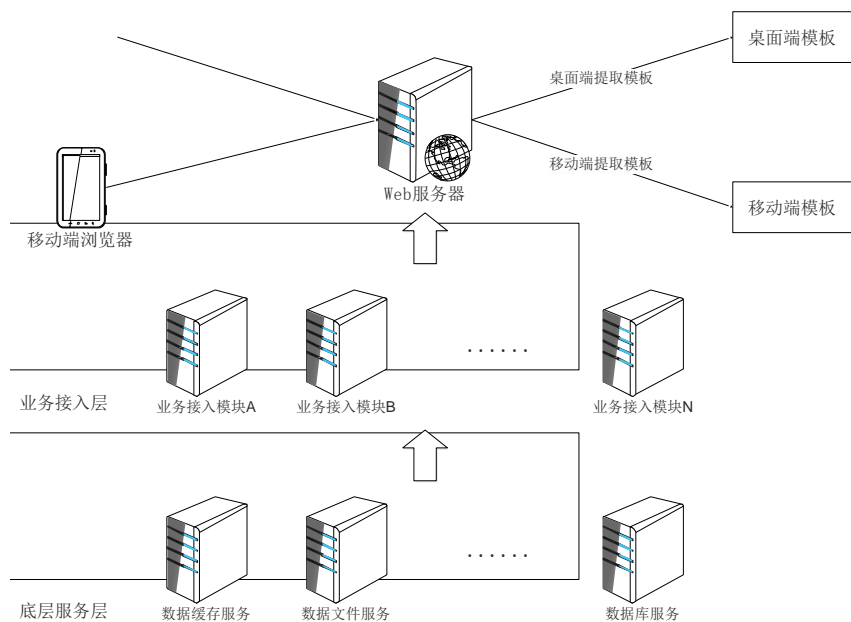


图 3-11 后端响应式内容渲染更加复杂的架构

## 📁 结构层媒体响应式

通过对不同开发模式中数据渲染思路的分析，我们基本解决了结构层 HTML 响应式所面临的主要问题。细节上，有一点需要重点强调：结构层媒体响应式的实现。根据统计，目前主要网站 60% 以上的流量数据来自图片，所以如何在保证用户访问网页体验不降低的前提下尽可能地降低网站图片的输出流量具有很重要的意义。

通常在我们手机访问网页时，请求的图片可能还是加载了与桌面端浏览器相同的大图，文件体积大，消耗流量多，请求延时长。媒体响应式要解决的一个关键问题就是让浏览器上的展示媒体内容尺寸根据屏幕宽度或屏幕分辨率进行自适应调节。当然这里提到的媒体主要是指图片，即我们需要根据浏览器设备屏幕宽度和屏幕的分辨率来加载不同大小尺寸的图片，避免在移动端上加载体积过大的资源，下面来看看前端图片响应式的几种常见解决方案。

### 1. 使用 Media Query 背景图片代替

前端结构层图片响应式一个常见的方案是使用背景图片引入来引入页面上的图片，并在 CSS 中通过 Media Query 来判断加载所需要的不同背景图片，这样浏览器就会根据浏览器设备的屏幕宽度或屏幕分辨率来加载不同的图片了。

```
.image {
  background-image: url('path/image/picture.jpg?w/1080/h/768.jpg');
  -webkit-background-size: 100% 100%;
  background-size: 100% 100%;
}

@media only screen and (max-width: 414px) {
  .image {
    background-image: url('path/image/picture.jpg?w/540/h/384.jpg');
  }
}

@media only screen and (max-width: 375px) {
  .image {
    background-image: url('path/image/picture.jpg?w/270/h/192.jpg');
  }
}

@media only screen and (max-device-pixel-ratio: 2) {
  .image {
    background-image: url('path/image/picture.jpg?w/270/h/192.jpg');
  }
}

@media only screen and (min-device-pixel-ratio: 2) {
  .image {
    background-image: url('path/image/picture.jpg?w/540/h/384.jpg');
  }
}
```

例如此处地址带有?w/1080/h/768、?w/540/h/384、?w/270/h/192 参数的图片分别表示大图片、中图片、小图片。在屏幕宽度小于 375 像素或屏幕分辨率小于 2 时, 使用小图片展示; 如果屏幕宽度在 375 像素到 414 像素之间或屏幕分辨率大于等于 2 时, 则使用中图片显示; 否则使用大图片显示。使用这种方法可以较好地解决移动端和桌面端不同浏览器下响应式图片的加载问题, 但是这里的图片由于是背景图片, 无法定义页面图片属性和描述内容, 不利于搜索引擎优化 (Search Engine Optimization, SEO), 也不能在图片加载失败时给予文字提示, 而且如果图片内容是动态生成的, 那么就需要修改标签元素的 style 属性来设置背景图, 显然这是不合理的。

由于高清屏的特性, 以 2 倍 Retina 高清屏的移动设备为例, CSS 中的 1px 是由  $2 \times 2$  个屏幕物理像素点来渲染的, 那么样式上的 border: 1px 在 Retina 高清屏下会渲染成 2 个物理像素宽度或高度的边框, 有时为了追求 1px 精准的还原, 不得不思考其他的方法来解决这个问题。实现 1px 边框的方式比较多, 通常可以设置元素 after 或 before 伪元素为 1px 内容, 并

使用 `transform: scaleY (1/devicePixelRatio)` 来进行单方向的缩放实现 1 个物理像素的边框或内容。对于字体,我们也可以设置 `transform: scale(.5)` 在浏览器中支持显示小于 12px 的文字。同时如果页面的内容因为使用高清屏而导致模糊,则需要使用 `-webkit-font-smoothing: antialiased` 来尝试修复。

## 2. Picture 标签元素

除了使用 Media Query 来实现图片响应式的展示外, W3C 已经有了一个用于实现响应式图形的草案,即新定义的 HTML5 标签 `<picture>`, 但因为它还只是草案,目前还没有较多支持的浏览器,因此只能期待在不久的未来我们能用上。尽管目前不支持,但其还是可以作为一种可选的实现方案, `<picture>` 元素标签是一个类似 `<img>` 展示图片的元素,但图片内容是由多个源图组成,并能根据屏幕的特性选择使用不同的图片,举例如下。

```
<picture width="500" height="500">
  <source media="(min-width: 640px)" srcset="large-1.jpg 1x, large-2.jpg 2x">
  <source media="(min-width: 320px)" srcset="middle-1.jpg 1x, middle-2.jpg 2x">
  <source srcset="small-1.jpg 1x, small-2.jpg 2x">
  
  <p>Accessible text</p>
  <noscript>
    
  </noscript>
</picture>
<!--
source: 表示<picture>的一个图片源;
media: 媒体查询,用于适配屏幕尺寸;
srcset: 图片链接, 1x 适应普通屏, 2x 适应高清屏;
<noscript/>: 当浏览器不支持脚本时的一个替代方案;
<img/>: 初始图片; 另外还有一个无障碍文本,类似<img/>的 alt 属性。
-->
<picture>
```

从这个例子来看,当用户浏览器在屏幕宽度大于 640 像素时,如果屏幕分辨率为 1 则使用 `large-1.jpg`,分辨率为 2 则使用 `large-2.jpg`;当屏幕宽度在 320 像素到 640 像素之间时,屏幕分辨率为 1 则使用 `middle-1.jpg`,分辨率为 2 则使用 `middle-2.jpg`;其他情况下,如果屏幕分辨率为 1 则使用 `small-1.jpg`,分辨率为 2 则使用 `small-2.jpg`。这样就可以根据不同浏览器屏幕特性来解决图片响应式的问题了。

目前由于大部分主流的浏览器还不支持 `<picture>` 元素,但它的原理是我们可借鉴的,所以就有了用于图片响应式处理 `<picture>` 元素的 Polyfill (Polyfill 是指使用第三方手段让浏览器支持浏览器原本并不支持的新特性) 思路——Picturefill。Picturefill 是 W3C 提供的最新的针

对响应式图片的设计方案，解析的过程主要如下：通过 JavaScript 脚本获取<picture>元素中的 Source 源以及 CSS Media Queries 规则，再根据浏览器的尺寸或分辨率信息将对应的图片路径赋值给<img>标签的 src 属性来加载不同的图片，从而兼容现有不支持<picture>元素的浏览器。相比之下，这种方式也为图片响应式提供了另一个可选的方案，但仍需要考虑 Picturefill 的性能解析问题，尤其是在移动端图片较多的页面，对应每个图片都需要去解析，可能会因为解析速度慢而阻塞其他页面脚本逻辑的执行。

这里要注意的是，浏览器端的 Polyfill 和 shim 差别很小，其实可以认为基本是没有区别的，都是为了解决旧的浏览器对于新特性的支持和兼容性问题。例如，es5-shim.js 是为了让旧的浏览器支持 ECMAScript 5 的特性；Picturefill 是为了让旧的浏览器支持解析 HTML5 的<picture>新标签。

### 3. 模板判断响应式图片

在前端渲染数据的开发模式下，使用前端模板进行判断渲染输出不同的图片是最简单、最直接的响应式图片实现方式。我们可以判断浏览器 userAgent 或检测是否高清屏 `let isRetina = window.devicePixelRatio > 1;` 来填充不同尺寸的图片地址到页面模板中。这种方法实质上和<picture>元素的 Polyfill 思路类似，但是更直接、更易理解实现。例如直接在前端渲染的模板中就可以如下使用。

```
{% if isMobile && isRetina %}
  
{% elseif isMobile %}
  
{% else %}
  
{% endif %}
```

这里前端模板首先判断浏览器是否为移动端浏览器，同时判断屏幕清晰度，若为移动端浏览器且为高清屏幕，则使用中图片；如果是移动端浏览器，但不是高清屏幕，则使用小图片；否则为桌面端浏览器使用大图片渲染。用这种方式处理起来就很直接了，当然这种方式除了能在前端模板数据渲染时使用以外，也能应用于后台或直出层页面模板的判断。所以在目前<picture>元素支持不成熟且不想用背景图片代替的方式前提下，使用模板来直接判断是很适用的方案。

### 4. 图片服务器判断输出内容

如果觉得在模板中使用判断的方式渲染不同的图片依然显得比较麻烦，我们也可以在图片

服务器上进行自适应修改。试想，如果将图片输出的判断逻辑放在后台图片静态服务器或内容分发网络（Content Deliver Network，CDN）上处理，就不用关心这些问题了，CDN 的基本思想是尽可能避开互联网上有可能影响数据传输速度和稳定性的环节，实现内容的快速、稳定传输。通常这种方案是通过浏览器访问服务器图片时带上浏览器的 `userAgent` 或 `URL` 参数等信息来实现的，服务器读取到这些信息后结合 `userAgent` 的不同浏览器特点输出不同大小的图片。总之，可以简单理解为将设备浏览器的判断放在图片服务器上实现，对于同一个图片 `URL` 使用什么尺寸的图片输出完全由图片服务器决定。

这是一个服务端的解决方案，优点是前端几乎不用做任何修改就可以实现按照不同的设备屏幕特点呈现不同大小的图片，因此可以快捷地应用于历史的项目迁移改造中，而且不会有兼容性问题。例如在前端图片的请求里可以带上请求的简单参数，这样服务器不仅可以自动输出对应的图片内容，而且还可以根据指定的参数进行优化格式输出。

```

```

### 3.7.3 表现层响应式

了解完结构层响应式，我们再来看一下表现层响应式的具体实现。这里至少要了解两个方面的内容：响应式布局和屏幕适配布局。响应式布局是根据浏览器宽度、分辨率、横屏、竖屏等情况来自动改变页面元素展示的一种布局方式，一般可以使用栅格方式来实现，实现思路有两种：一种是桌面端浏览器优先，扩展到移动端浏览器适配；另一种则是以移动端浏览器优先，扩展到桌面端浏览器适配。由于移动端的网络和计算资源相对较少，所以一般比较推荐从移动端扩展到桌面端的方式进行适配，这样就避免了在移动端加载冗余的桌面端 `CSS` 样式内容。而屏幕适配布局则是主要针对移动端的，由于目前移动端设备屏幕大小各不相同，屏幕适配布局是为了实现网页内容根据移动端设备屏幕大小等比例缩放所提出的一种布局计算方式。我们先来看下页面的响应式布局。

#### 👉 响应式布局

响应式布局的思路比较直接，一般是通过栅格系统来解决百分比方式布局。例如我们希望一些元素的宽度在桌面端浏览器上按一定比例布局，而在移动端统一占用一行，那么就可以采用如下方式。

```
.row {  
  width: 100%;  
}  
.row .col-1 {
```



```

        width: 8.333333333333%;
    }
    .row .col-2 {
        width: 16.66666666667%;
    }

    /* ...比较多, 此处省略 */
    .row .col-12 {
        width: 100%;
    }

    /* 屏幕宽度小于 414px 的情况 */
    @media only screen and(max-width: 414px) {
        .row .col-1, .row .col-2, .row .col-3, .row .col-4, .row .col-5, .row .col-6, .row
        .col-7, .row .col-8, .row .col-9, .row .col-10, .row .col-11 {
            width: 100%;
        }
    }

    /* 竖屏的情况 */
    @media screen and (orientation: portrait){
        ...
    }

    /* 横屏的情况 */
    @media screen and (orientation: landscape){

    }

```

栅格化布局通常会将屏幕宽度等分成多个固定的栅格（以 12 格为例），在屏幕宽度大于 414px 的情况下，每个栅格的元素使用宽度为按照列数定义父元素的百分比宽度；在屏幕宽度小于 414px 的情况下，判断是移动端设备或浏览器宽度较窄，所有的栅格元素宽度设置为 100%，这就避免了移动端屏幕上容器宽度按百分比计算较小的问题。此外也可以根据移动端设备横屏或竖屏的情况添加特殊的样式规则。

```

<div class="row col-1"></div>
<div class="row col-2"></div>
<div class="row col-4"></div>
<div class="row col-8"></div>
<div class="row col-12"></div>

```

图 3-12 为上面 HTML 结构内容在浏览器窗口分别大于 414px 和小于 414px 的情况，这样实现就保证了移动端浏览器上元素布局的自适应显示，让移动端内容的布局更加合理清晰。

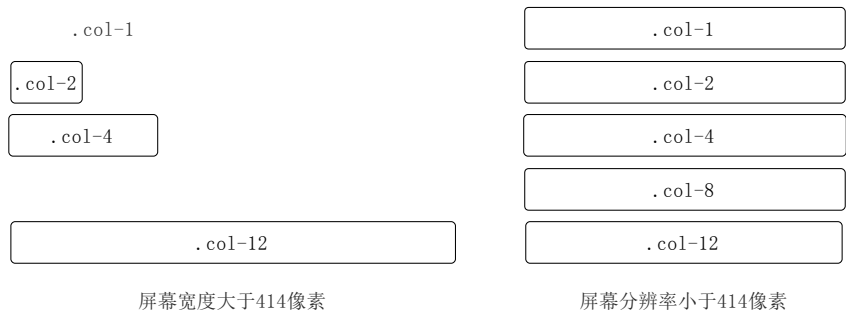


图 3-12 栅格化布局展示效果

### 👉 屏幕适配布局

屏幕适配布局是在移动端解决内容按照不同屏幕大小自动等比例缩放的一种布局计算方式。屏幕适配布局和响应式布局是不同的，一般只在移动端使用。通常在移动端页面上，首先为了固定浏览器对 HTML 文件的渲染，会在 HTML 的<head>里面加上下面一段<meta>声明来控制页面使用移动端浏览器展示并保持内容不缩放。

```
<meta name="viewport" content="width=device-width,initial-scale=1, maximum-scale=1, user-scalable=no" />
```

这里通过<meta>控制页面的不缩放和我们屏幕适配布局根据不同屏幕大小自动缩放的概念是不同的。<meta>中 viewport 控制的缩放是在屏幕宽度确定后浏览器的视窗内容不随用户的操作而缩放，而屏幕适配布局的自动缩放是在屏幕宽度不确定的情况下页面元素展示内容与宽屏大小保持比例不变。不然，可能我们在小屏移动设备上显示正常的字体在大屏手机上显示视觉上就可能很小了。

以 320px 宽度的移动设备屏幕和 414px 宽度的移动设备屏幕比较来看，图 3-13 是使用模拟器在屏幕宽度为 320px 的浏览器和屏幕宽度为 414px 的浏览器中打开同一个页面的情况，页面中宽度为 160px 的容器在 320px 宽度的屏幕下面占总宽度的 50%，而在 414px 宽度的屏幕下面显然没有占到 50%，而且不同屏幕下面能显示同等大小的字体个数也是不相同的，屏幕宽度为 414px 的屏幕下面一行可以显示更多的字数，这样就导致了页面内容展示在不同移动端屏幕上不一致。对于这类屏幕适配布局问题，我们通常有两种处理方法，即依据 HTML 中<html>标签元素的 zoom 属性缩放和根据 REM 自适应方案实现等比例缩放。

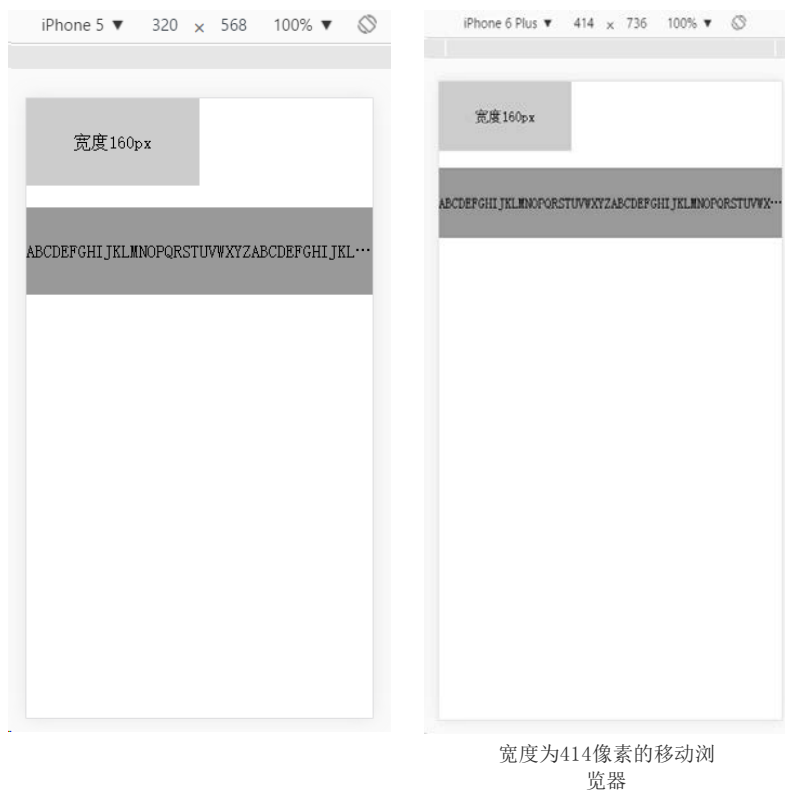


图 3-13 不同大小屏幕下页面显示效果

### 1. zoom 属性控制方案

如果希望在 320px 宽度屏幕上显示的内容在 414px 的宽度屏幕上进行等比例自动放大，可以考虑使用元素 CSS 的 zoom 属性来尝试解决。例如我们可以设置<html>或<body>标签的 CSS 属性为“zoom: 1.29375”，其中  $1.29375 = 414/320$ ，即将 zoom 的值按照屏幕宽度的 320 分之一计算，然后赋给 HTML 文档的<html>或<body>元素样式就可以了。这种情况我们使用 414px 宽度的移动端屏幕打开同一个页面看到的内容比例和在 320px 宽度移动端浏览器打开看到的内容比例是一致的。使用 JavaScript 就可以按如下方式来设置 body 的 zoom 属性。

```
document.body.style.zoom = screen.width/320;
```

如图 3-14 所示，通过实践，加上“zoom: 1.29375”属性后，在宽度为 414px 的屏幕上的内容显示比例和宽度为 320px 的屏幕上保持一致。这种方式实现很简洁，但是这个 zoom 值通常需要通过 JavaScript 计算得出，如果 JavaScript 在页面渲染之后完成，则会出现页面内容全部重排的情况，所以我们需要尽量在页面文档开始的地方给<html>或<body>设置 zoom 属性。

但是这样处理之后，其他屏幕下面的缩放参考尺寸必须全部按照这个固定的比例来缩放，显得不太灵活。



图 3-14 使用 zoom 属性缩放后展示效果

## 2. REM 屏幕适配方案

REM 方案目前应用广泛。我们知道标签元素的 CSS 大小尺寸的表达单位主要有像素 px、相对父元素大小百分比%、相对于当前对象内文本字体 font-size 的大小 em、相对于文档根元素 font-size 的大小 rem（当然还有新的 vh、vw、vmax、vmin，此处先不做讨论，大家有兴趣可以去了解，其实是解决屏幕适配更好的备选方案，但是目前兼容性不是太好）。

所以在 REM 方案中，我们如果给 HTML 根元素一个根据屏幕自动调整的 font-size，页面上所有元素的尺寸全部以 rem 为单位，无论屏幕宽度怎样变化，页面的内容和屏幕的比例将始终是不变的，这样就可以实现页面内容根据屏幕来自动缩放了。

这里定义 1rem 大小的方案有很多，而且计算得到的 1rem 对应的像素宽度结果也都不一样，

例如有人这样来计算。

$$1\text{rem} = \text{屏幕宽度} \times \text{屏幕分辨率} / 10$$

上述方法得到的 `1rem` 恰好是屏幕宽度的 10%，所有的尺寸布局都相当于完全使用百分比来布局，这样就可以适应几乎所有的屏幕。

或者我们通常以某个屏幕宽度的设计稿为基准（例如 320 像素宽度）进行缩放。

$$1\text{rem} = \text{屏幕宽度} / 320 \times 10$$

这样 `1rem` 在宽度为 320px 的屏幕上表示的是 10px，按照这个基准，`1rem` 在其他屏幕上页面的显示比例和在 320px 宽度的屏幕上显示比例将是一致的。

在上面的例子中，如果 `<html>` 标签元素的 `font-size` 为 10px，此时将页面第一个容器块的宽度设置为 `width: 16rem`，那么在屏幕宽度为 414 px 的浏览器上，`<html>` 元素的 `font-size` 设置为  $414/320 \times 10\text{px}$ ，容器块的宽度将自动填充为屏幕宽度的一半。同样，页面文字的设置也可以完全使用 `rem` 为单位来实现屏幕适配。这里 `rem` 通常也是通过 JavaScript 的计算得出的。除此之外，我们也可以像下面这样通过 Media Query 直接对常见的几种屏幕宽度的 `<html>` 元素标签来进行设置，避免使用 JavaScript 来计算。

```
@mixin queryWidth( $min , $max ){
  @if $min == -1{
    @media screen and ( max-width: $max+px ) {
      html{
        font-size: ( ($max+1) / 320 ) * 10px;
      }
    }
  } @else if $max == -1{
    @media screen and ( min-width: $min+px ) {
      html{
        font-size: ( $min / 320 ) * 10px;
      }
    }
  } @else{
    @media screen and ( min-width: $min+px ) and ( max-width: $max+px ) {
      html{
        font-size: ( $min / 320 ) * 10px;
      }
    }
  }
}

@include queryWidth(-1,319);
```

```
@include queryWidth(320,359);
@include queryWidth(360,374);
@include queryWidth(375,383);
@include queryWidth(384,399);
@include queryWidth(400,413);
@include queryWidth(414,-1);
```

通过 SASS 语法的计算设定很好地解决了不同大小屏幕下内容和屏幕比例不同的问题，其实怎样来计算 1rem 的大小并没有最佳的方案，任何一种方案都是合理的，具体需要根据设计稿大小和需求来确定。而且需要注意的是，REM 方案目前是移动端上很成熟和使用最广泛的屏幕适配方案，因为相对于 zoom 属性的设置，更加灵活可控，可以结合不需要缩放的 px 单位一起使用，所以建议在移动端尽量都用 REM 来做屏幕适配布局。

### 3.7.4 行为层响应式

在页面的响应式设计中，行为层脚本也是需要根据浏览器环境来执行不同逻辑的。在 3.7.2 节中我们也提到了一些，和结构层类似，行为层的响应式同样分为 JavaScript 内容在前端引入和在后端引入这两种情况。对于前一种情况，我们主要可以通过设备浏览器环境判断来异步加载不同的 JavaScript 脚本，这里不做过多讲解。

```
if(isMobile){
  require.async(['zepto', './mobileMain'], function($, main){
    main.init();
  });
}else{
  require.async(['jQuery', './main'], function($, main){
    main.init();
  });
}
```

当然如果是在后端直出层渲染不同脚本内容就更简单了，通过设备浏览器的判断在输出模板中引入不同的脚本资源路径就可以了。

```
{% if isMobile %}
  <script src="path/mobile/main.js"></script>
{% else %}
  <script src="path/pc/main.js"></script>
{% endif %}
```

这样便有效保证了桌面端浏览器和移动端浏览器只加载自己需要的脚本资源，执行对应的逻辑，同时避免了冗余的资源文件下载。

## 3.8 本章小结

这一章中，我们围绕前端的三层结构展开介绍了 JavaScript、HTML 和 CSS 标准的演进与主要特性的变化，以及它们在现代开发中的实际应用。就响应式网站的设计与实现，笔者结合自己的实践经验进行了较全面的原理性剖析和总结，介绍了响应式网站实践需要考虑到的问题与解决方案。掌握这些将有利于我们把握住前端技术的基础和应用方法，这也是前端学习中最核心的部分。下一章中，我们将为大家继续讲解前端 DOM 交互框架设计的内容，主要以前端框架的演进为主线来讲述 DOM 交互模式的变化，同时向大家讲解 DOM 交互方式的设计原理和思路。

# 第4章

## 现代前端交互框架

Web 前端页面的开发避免不了与 DOM 的交互操作。自前端技术出现后，页面的结构越来越复杂，用户的操作交互越来越多，对应的 DOM 交互也越来越频繁。为了提高开发效率，我们常常借助 DOM 交互框架来简化页面的开发工作。经过 Web 前端这些年的发展，用于 DOM 交互的框架越来越多，设计思路也不尽相同。总结来说，前端框架一次次变化，从提升效率的阶段，慢慢走向改善性能的阶段。这章中我们就一起来看看那些备受关注的前端框架，了解一下这些前端框架的变化究竟给我们带来了什么。

### 4.1 直接DOM操作时代

前端 HTML 结构是浏览器中承载互联网内容数据的主要载体，用户对数据的处理和展示都可以通过 HTML 来体现。而对于开发者来说，所有数据内容都可以说是通过 DOM 结构来组织和展示的。数据的处理和操作的核心其实就是 DOM 的处理和操作，即便是今天，所有前端 JavaScript 框架最终要解决的仍然是如何实现高效、高性能 DOM 交互操作的问题。

我们先回到 Web 前端的起始阶段，那时候其实是没有页面 DOM 交互的，一般就是一个静态黄页，即把一个静态的文本放到一个连接外网的服务器目录下，供用户通过浏览器来连接访问。用户除了点击页面跳转外基本不能有任何复杂的操作，网页上的内容也不能动态更新，只能通过开发者修改静态文件来改变。早期的网站页面作为一个单纯向用户传递信息和共享数据的方式存在。

很快，使用数据库技术便可以将数据库中的数据记录读取出来并展示给用户，此时用户已经可以在页面上进行一些简单的操作了，例如表单提交、文件上传等，这一阶段我们可以称为 Web 前端的早期 DOM 交互时代。在该阶段，用户仍以单向订阅、获取和接受网站信息内容为



主，可以进行刷新式的页面提交等操作。当然今天来看这些技术就有点久远了，尤其在互联网如此高速发展的前提下就显得更遥远。

本书一开始便向读者介绍了前端技术出现的背景和缘由，从根本上来说也可以认为前端技术的出现是为了将 DOM 的交互操作从整个 Web 站点开发中独立出来，进而进行更加高效的管理。随着 AJAX（Asynchronous JavaScript And XML，异步 JavaScript 和 XML）技术的出现，前端页面上的用户操作越来越多，越来越复杂，以前的很多用户请求都可以通过 AJAX 无刷新的操作来完成，通常 AJAX 的流程为用户使用 XMLHttpRequest（或 ActiveX）创建 HTTP 请求来获取服务端的数据或一段 HTML 结构内容，请求成功后在页面上进行增加、修改、删除 DOM 元素的操作。

要实现对 DOM 元素的交互操作，就不得不提到 DOM API，DOM API 提供了浏览器上进行 DOM 对象树交互操作的一系列原生 JavaScript 方法，例如 getElementById、createElement、createDocumentFragment、appendChild 等。包括 HTML5 新增加的 DOM API 与 AJAX 的 API 在内，这些 API 可以分成以下几种类型：节点查询型、节点创建型、节点修改型、节点关系型、节点属性型和内容加载型。

表 4-1 列出的是目前浏览器常见的 DOM API 方法。使用这些基本的 API 我们就可以进行前端页面上几乎任何 DOM 的查询和网络请求操作了，早期页面相对比较简单，没有太多的操作内容，使用这些原生的 API 就可以满足开发需要了。例如我们要创建一个数字列表，每个列表项都有固定 id 和 data-id，我们就可以按照如下代码通过基础的 DOM API 实现。

表 4-1 常见 DOM API 举例

类 型	方 法
节点查询型	getElementById 、 getElementsByName 、 getElementsByClassName 、 getElementsByTagName 、 querySelector、querySelectorAll
节点创建型	createElement、createDocumentFragment、createTextNode、cloneNode
节点修改型	appendChild、replaceChild、removeChild、insertBefore、innerHTML
节点关系型	parentNode、previousSibling、childNodes
节点属性型	innerHTML、attributes、getAttribute、setAttribute、getComputedStyle
内容加载型	XMLHttpRequest、ActiveX

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
```

```
</head>
<body>
  <div>
    <h2>创建列表</h2>
    <ul id="menu"></ul>
  </div>
  <script>
    let menuUl = document.getElementById('menu');
    let fragment = document.createDocumentFragment();
    for (let i = 0; i < 5; i++) {
      let li = document.createElement('li');
      li.innerHTML = i;
      li.id = i;
      li.setAttribute('data-id', i);
      fragment.appendChild(li);
    }

    // 一次性将文档碎片内容插入到 DOM 中
    menuUl.appendChild(fragment);
  </script>
</body>
</html>
```

需要注意的是,这里 DOM 的 property 和 attribute 是有区别的。property 通常是指 DOM 元素对象的属性,例如 style; attribute 是指 HTML 标签的文本标记属性,一般是可见的,如自定义的 data-id 属性。再如一个元素可以设置 style property 和 style attribute,但是 style property 是 DOM 元素固有的,而 style attribute 则不一定有,需要在 HTML 结构中设置指定。推荐使用 createDocumentFragment 来代替 createElement 创建节点内容,因为 createDocumentFragment 可以将多个文档内容片段先缓存起来,最后一次性插入到 DOM 中,而 createElement 每次创建节点都需要插入到 DOM 中,所以 createDocumentFragment 可以减少 DOM 操作次数,提高效率。

内容加载型 API 的实现方式代码如下。

```
function urlGet(url) {
  let xmlhttp;

  // 创建 xmlhttp
  if (window.XMLHttpRequest) {
    xmlhttp = new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
  }
}
```

```
}

// 发送 xmlhttp 请求
xmlhttp.open("GET", url);
xmlhttp.onreadystatechange = function() {
    if ((xmlhttp.readyState == 4) && (xmlhttp.status == 200)) {
        alert('success');
    } else {
        alert('fail');
    }
}
xmlhttp.send(null);
}
```

通过使用这些 API，我们基本可以完成前端网页中的任何操作。但随着网站应用的复杂化，使用这些原生的 API 开发就显得比较低效而且不易管理。因为想对这套 API 进行必要的封装来提高调用效率，所以 jQuery 这个典型的 DOM 交互框架就应运而生了，一起出现的还有 prototype 和 motools，但是后两者一开始就实现了类的设计模式而且 API 的使用不太方便，因此没有被广泛使用。jQuery 以其简单友好的 API 和书写方式很快得到了广大开发者的认可和使用。

移动端开发类似 jQuery 的框架，以 zepto 为主，可以认为它是一个简化版的 jQuery，其常用的 API 和 jQuery 完全相同。

那么，我们再来看一下 jQuery 在那个时代甚至现在到底帮我们解决了什么问题呢？为什么能够受到这么多人的青睐？大概一想，jQuery 帮我们解决的问题太多了，添加了丰富的 API 方法的封装和网络操作，这些都是浏览器本身没有的，极大地提升了开发效率。但仔细一想，jQuery 处理的问题似乎又没有那么难，jQuery 只是帮我们解决了使用基础 DOM API 进行前端开发遇到的一些问题。再回到 DOM 原生 API 的几种类型上，可以发现 jQuery 基本帮我们进行了上面 DOM 的六类 API 的封装操作，方便了开发者对这几类 API 进行调用。

表 4-2 所示为 jQuery 部分常见的方法 API，这些封装的 API 大大简化了我们的代码开发步骤，提高了开发效率。原来使用基础 DOM API 开发的方式就可以变成 jQuery 更简单的方式来实现了。

表 4-2 常见 jQuery API 举例

类 型	方 法
节点查询型	\$(selector)、find()等
节点创建型	\$( selector)、clone()等
节点修改型	html ()、replace()、remove() 、append()、before()、 after()等

续表

类 型	方 法
节点关系型	parent()、siblings()、closest()、next()、children()等
节点属性型	attr()、data()、css()、hide()、show()、slideDown()、slideUp()、animate()等
内容加载型	ajax()、get()、post()等

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
</head>
<body>
  <div>
    <h2>创建列表</h2>
    <ul id="menu"></ul>
  </div>
  <script src="./jQuery.js"></script>
  <script>
    let htmlArr = [];
    for (let i = 0; i < 5; i++) {
      htmlArr.push(`<li id="${i}" data-id="${i}">${i}</li>`);
    }
    $('#menu').append(htmlArr.join(''));
  </script>
</body>
</html>
```

相对来说，开发的代码简洁了很多，对于网络加载型 API 而言，也可以使用非常便捷的方式实现，读者可以和基础 API 实现的方式进行对比。

```
function urlGet(url) {
  $.ajax({
    url: url,
    type: 'get',
    success(){
      alert('success');
    },
    error(){
      alert('fail');
    }
  });
}
```

虽然经过了几个大版本的迭代，目前 jQuery 被使用最为广泛的版本仍然是 1.x 版本。目前为止 jQuery 提供的 API 已经很完善了，jQuery 1.x 版本帮我们解决了许多问题。

- 简化选择器。可以使用 `$('#id .class-a')`、`find()` 这种简短的形式进行组合查询，帮助我们快速找到所有满足条件的 DOM 元素，并自动在原型链上为返回的对象添加常用的操作方法。相比之下，原生的方法实现组合查询就比较复杂了。
- DOM 操作方法。扩展实现了如 `html()`、`append()`、`remove()`、`hide()`、`animate()` 等多种类型的 DOM 操作方法，让 DOM 的交互和修改更加简单。
- AJAX。实现了对 XMLHttpRequest 和 ActiveX 的统一封装，使 AJAX 网络请求的调用更加方便。

AJAX 跨域请求时默认不会带有浏览器端 Cookie 信息，需要在请求头部加上 `xhrFields: { withCredentials: true }` 才能将 Cookie 信息正常带到请求中发送给服务器。

- 事件绑定统一处理。除了 DOM API 的封装，jQuery 也对 DOM 的部分功能做了封装，主要添加了 `on` 等方法来统一处理事件，包括事件绑定和事件代理等。

```
// BOM API 实现事件绑定
addEvent(element, type, fn) {
  // 兼容性判断
  if (element.addEventListener) {
    element.addEventListener(type, fn, false);
  } else if (element.attachEvent) {
    // 通常 IE 上回调函数的 this 不指向当前 element，所以绑定时使用 fn.call(element) 来
    // 绑定元素
    element.attachEvent('on' + type, function() {
      fn.call(element);
    });
  } else {
    element['on' + type] = fn;
  }
}

// jQuery 事件绑定
$(element).on(type, fn);
```

- 延时对象。较新的 jQuery 版本添加了 `$.Deferred` 对象来处理异步回调嵌套的问题。`$.Deferred` 的实现借鉴了异步处理的 Promise/A 规范，但并没有完全遵循 Promise/A 规范。一般执行 `resolve` 时，`$.Deferred` 对象会一次性执行 `done` 中全部的方法。

```
let $defer = $.Deferred();

$defer.done(function() {
  console.log('A');
```

```
}).done(function() {  
    console.log('B');  
}).done(function() {  
    console.log('C');  
}).done(function() {  
    console.log('D');  
});  
$defer.resolve();
```

我们再来看一个异步的例子，理解一下\$.Deferred 是怎样控制多个异步函数执行的。

```
const fn1 = function() {  
  
    // 声明$defer 来控制执行流程，但不调用$defer 时将不会继续执行返回。这样我们就可以用来封装  
    // AJAX 等方法以保证请求的有序性了  
    let $defer = $.Deferred();  
    setTimeout(function() {  
        console.log('A');  
        $defer.resolve();  
    }, 3000);  
    return $defer;  
}  
  
const fn2 = function() {  
    let $defer = $.Deferred();  
    setTimeout(function() {  
        console.log('B');  
        $defer.resolve();  
    }, 2000);  
    return $defer;  
}  
  
const fn3 = function() {  
    let $defer = $.Deferred();  
    setTimeout(function() {  
        console.log('C');  
        $defer.resolve();  
    }, 1000);  
    return $defer;  
}  
  
const fn4 = function() {  
    setTimeout(function() {  
        console.log('D');  
    }, 0);  
}  
  
fn1().then(fn2).then(fn3).then(fn4);
```

- 兼容性实现。例如 jQuery 实现 on 事件绑定和 Ajax 封装时充分考虑了不同浏览器的差异性，并做统一处理，做到兼容性问题对开发者的透明。

总而言之，jQuery 主要实现了选择器、DOM 操作方法、事件绑定封装、AJAX、Deferred 这五个方面的封装和常见的兼容性问题的处理。除此之外，我们还可以基于 jQuery 扩展更多的方法功能来提高业务开发效率，例如 Cookie 和 localStorage 操作的封装、上传文件、二维码自动生成等都可以基于 jQuery 的大框架扩展实现。

如果你因为项目的历史原因还在使用 jQuery，那么我想要给些建议。想要高效地使用 jQuery，可以参考以下的优化建议和原则：（1）尽可能使用 id 选择器进行 DOM 查询操作，不要使用组合选择器；（2）缓存一切需要复用的 jQuery DOM 对象，使用 find() 子查询；（3）不要滥用 jQuery，尽量使用原生的代码代替；（4）尽可能使用 jQuery 的静态方法；（5）使用事件代理，不要直接使用元素的事件绑定；（6）尽量使用较新的 jQuery 版本；（7）尽可能使用链式写法来提高编程效率和代码运行效率。

为什么要提到上述几点，是因为很多人仍然不注意 jQuery 的高效编程原则，容易犯常见错误，包括一些国内一线企业里面工作时间很长的开发者仍在犯这些错误，这样做虽然不会影响业务的功能开发，但是做完并做好一件事情应该成为我们的另一个目标。希望你身上没出现过这些问题。

图 4-1 是目前前端主流 DOM 交互式页面加载的基本流程，浏览器开始加载页面 HTML，页面 HTML 加载完成后请求页面脚本加载，脚本加载完成再执行数据请求，最后进行数据渲染 DOM 操作和事件绑定。

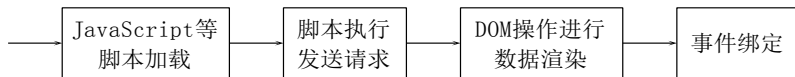


图 4-1 主流前端页面加载模式

随着 AJAX 技术的盛行，SPA（Single Page Application，单页面应用）应用开始被广泛认可。SPA 的思路是将整个应用的内容都在一个页面中实现并完全通过异步交互来根据用户操作加载不同的内容。这样，使用 jQuery 直接进行 DOM 交互的开发方式就显得不易管理。例如当 SPA 页面上的交互和异步加载的内容很多时，按照之前的思路，我们需要在每一次数据请求后进行数据渲染和事件绑定，用户操作后进行另一部分内容的请求和事件绑定，后面以此类推。当所有异步页面全部调用完成，页面上的绑定将变得十分混乱，各种元素绑定，渲染后的视图内容也不清晰，同时需要声明不同变量保存每次异步加载时返回的数据对象，因为页面交互需要保

留这些数据进行操作，最终的项目代码可能会乱成一锅粥。

在这种情况下，我们就很需要一个能自动管理页面上这些 DOM 交互操作的机制，这时或许就可以考虑一下使用 MV\* 的交互模式了。

## 4.2 MV\*交互模式

### 4.2.1 前端MVC模式

上节中我们讲到，通过 DOM 交互框架已经可以比较高效地处理 DOM 操作和事件绑定等问题了。这种高效的方式带来了效率上的提升，但随着页面结构和交互复杂性的提升，仅靠这种方式会增加维护管理的难度。随着 AJAX 技术的盛行，SPA 应用开始被广泛使用。上一节最后也提到，用这种直接的方式进行 SPA 的开发和维护是比较麻烦的。为了解决这个问题，通常将页面上与 DOM 相关的内容抽象成数据模型、视图、事件控制函数三部分，这就有了前端 MVC (Model-View-Controller) 的设计思路。

MVC 可以认为是一种开发设计模式，其基本思路是将 DOM 交互的内容分为数据模型、视图和事件控制函数三个部分，并对它们进行统一管理。Model 用来存放请求的数据结果和数据对象，View 用于页面 DOM 的更新与修改，Controller 则用于根据前端路由条件（例如不同的 HASH 路由）来调用不同 Model 给 View 渲染不同的数据内容。常用页面路由的实现也很简单，代码如下，其主要思路是让 URL 地址内容匹配对应的字符串然后进行相应的操作。

```
const router = {
  get(match, fn){
    let url = location.href,
        routeReg = new RegExp(match, 'g');
    if(routeReg.test(url)){
      fn();
    }
    return this;
  }
}

router.get('#index', function(){
  _loadIndex(); // 注册 hash 含有#index 的路由执行对应的操作
}).get('#detail', function(){
  _loadDetail(); // 注册 hash 含有#detail 的路由执行对应的操作
});
```

另外我们也可以使用 HTML5 的 pushState 来实现路由。history.pushState (state,



`title, url`) 方法可以改变当前页面的 `url` 而不发生跳转, 并将不同的 `state` 数据和对应的 `url` 对应起来。如果页面显示的内容是根据不同的数据状态来自动完成的, 这样根据 `state` 的内容来加载不同的组件就很有用了。

```
history.pushState({page: 'A'}, 'page A', 'a.html');
console.log(history.state); // {page: 'A'}对象

history.pushState({page: 'B'}, 'page B', 'b.html');
console.log(history.state); // {page: 'B'}对象

history.pushState({page: 'C'}, 'page C', 'c.html');
console.log(history.state); // {page: 'C'}对象
```

这里访问不同的 `URL` 地址 `a.html`、`b.html`、`c.html`, 页面并不会发生跳转刷新, 而是改变了当前的 `history.state` 内容, 我们使用 `history.state` 数据的改变来动态改变页面 `DOM` 的内容这种方式来实现 `SPA` 就很方便了。

使用路由后, 如果将 `SPA` 中的每个路由或用户操作加载的页面内容都看成是一个组件, 那么之前的做法是每个组件独立完成各自的数据请求操作、渲染和数据绑定, 一旦组件多了, 每个组件自行处理就会造成逻辑混乱。到了 `MVC` 里面, 所有的组件数据请求、渲染、页面逻辑操作都分别使用 `Model`、`View`、`Controller` 来注册调用。通俗地讲, 就像是组件交出了自己的控制权给统一的控制对象来调用一样。

如图 4-2 所示, 前端应用页面加载完成后, 根据不同的用户操作, 页面会执行不同的响应。例如用户操作或 `URL` 哈希改变时会去调用与之对应的 `ControllerA` 方法, `ControllerA` 方法会请求获取对应的数据 `ModelA`, `ModelA` 很可能是前端 `AJAX` 请求返回的一个接口数据, 然后将 `ModelA` 传递给视图模板 `ViewA` 并将最终内容渲染到浏览器中, 这样就完成了用户的一次交互操作, 用户下一次操作的过程也是一样的。同时如果用户的操作需要进行 `DOM` 结构修改, 那么会传入到 `Controller` 中, 通过 `Controller` 获取数据并控制 `View` 的更新。对其中的一个组件 `A` 的实现操作代码如下。

```
<div id="A" onclick="Controller.A.event.change"></div>

let Controller = {}, Model = {}, View= {};

View['A'] = function(data){
  let tpl = '<input id="input" type="text" value="{{text}}"><span id="showText">
{{text}}</span>';

  // 调用模板渲染数据获取 HTML 片段
  let html = render(tpl, data);
```

```
document.getElementById('A').innerHTML = html;
};

Model['A'] = {
  text: 'ViewA 渲染完成'
};

Controller['A'] = function(){

  View['A'](model['A']);

  // 用户操作一般通过改变 Hash 完成，并触发 Controller 来改变 Model 和 View
  $('window').on('hashchange', function(){
    model['A'].text = location.hash;
    View['A'](model['A']);
  });

  // 点击事件可以直接触发 Controller 改变 Model 并重新渲染 View
  self.event['change'] = function(){
    model['A'].text = '新的 ViewA 渲染完成';
    View['A'](model['A']);
  };
};
```

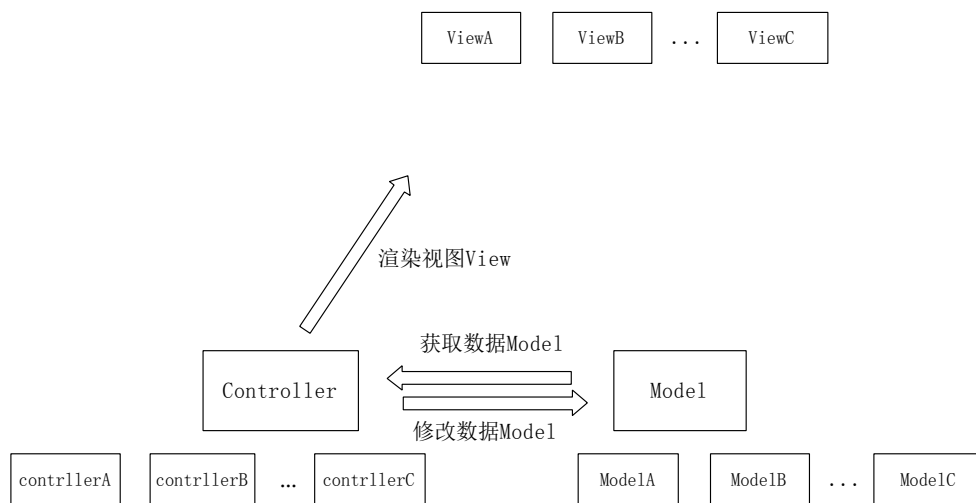


图 4-2 MVC 模式组件结构示意图

为了方便大家理解，这里采用了一个最直接的实现方式，而且没有对 A、B、C 组件进行单独管理，而是将它的调用内容分成 Controller、Model、View 并放在一起统一管理，相互之间的调用是直接进行的。成熟的 MVC 框架一般是通过事件监听或观察者模式来实现的，这里只是

为了让读者们更好地理解 MVC 的原理。当然这样的组件如果多了也比较混乱。所以通过改进可以将页面分成不同的小模块来处理，同时考虑到代码复用性，因此可以使用继承的方式来定义这三个组件，继续以 A 组件为例，我们一般看到的主流 MVC 框架的组件定义代码如下。

```
<div id="A" onclick="A.event.change"></div>

// 可能有一个公用的 Component 基类
let component = new Component();

let A = component.extend({
  $el: document.getElementById('A'),
  model: {
    text: 'ViewA 渲染完成'
  },

  view(data){
    let tpl = '<input id="input" type="text" value="{{text}}"><span
id="showText">{{text}}</span>';

    // 调用模板渲染数据获取 HTML 片段
    let html = render(tpl, data);
    this.$el.innerHTML = html;
  },

  controller (){
    let self = this;

    // 调用 model 数据传入 view 中渲染内容
    self.view(self.model);

    // 用户操作一般通过 Hash 来触发 Controller 改变 Model 和 View
    $('window').on('hashchange', function(){
      self.model.text = location.hash;
      self.view(self.model);
    });

    // 点击事件可以直接触发 Model 改变并重新渲染 View
    self.event['change'] = function(){
      self.model.text = '新的 ViewA 渲染完成';
      self.view(self.model);
    };
  }
});
```

尽管这里写法不太一样，但实现的功能和上面一段代码是相同的。当 Model 或 View 复杂后，我们也可以考虑将 Model、View、Controller 拆分成不同文件导入引用。这段代码就是 MVC

基础原型的设计和实现，需要注意的是，用户操作引起的 DOM 修改操作主要是通过 Controller 来直接控制的，但是 Controller 只进行修改操作指令的分发，数据的渲染一般是在 View 层来完成。

4.2.2 前端MVP模式

MVP（Model-View-Presenter）可以跟 MVC 对照起来看。和 MVC 一样，M 就是 Model，V 就是 View，而 P 代表 Presenter，它与 Controller 有点相似，但不同的是，用户在进行 DOM 修改操作时将通过 View 上的行为触发，然后将修改通知给 Presenter 来完成后面的 Model 修改和其他 View 的更新，而 MVC 模式下，用户的操作是直接通过 Controller 来控制的。Presenter 和 View 的操作绑定通常是双向的，View 的改变一般会触发 Presenter 的动作，Presenter 的动作也会改变 View。

图 4-3 为 MVP 的执行流程，Model 和 View 是不直接发生关系的，所有的逻辑调用数据 Model 和渲染视图 View 模板都在 Presenter 里面完成，同时用户在 View 层操作的改变会反馈到 Presenter 然后改变 Model 并渲染新的 View。因此，上一段代码通过 MVP 模式的实现如下。

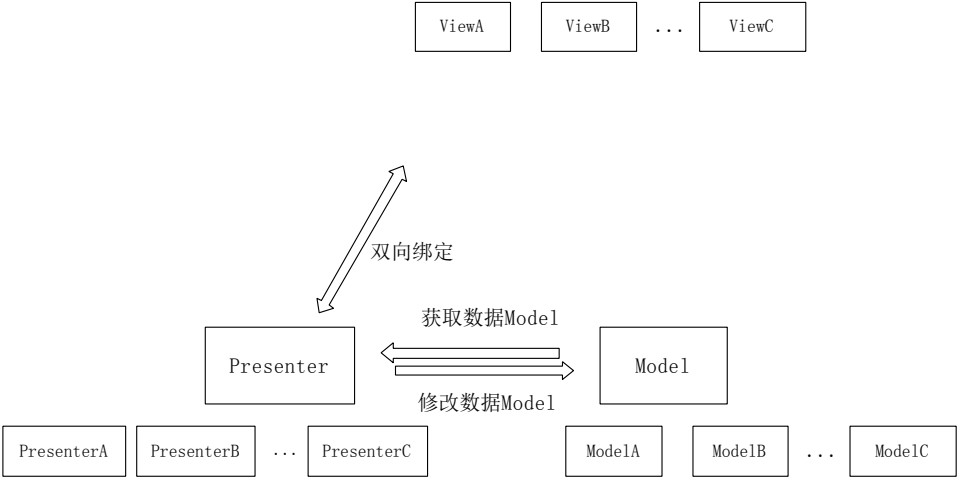


图 4-3 MVP 模式组件结构示意图

```
<div id="A" onclick="A.event.change"></div>
// 可能有一个公用的 Component 基类
let component = new Component();

let A = component.extend({
  $el: document.getElementById('A'),
```

```

model: {
  text: 'ViewA 渲染完成'
},

view: '<input id="input" type="text" value="{{text}}"><span
id="showText">{{text}}</span>',

presenter(){
  let self = this;

  // 调用模板渲染数据获取 HTML 片段
  let html = render(self.view, self.model);
  self.$el.innerHTML = html;

  // View 上的改变将通知 Presenter 改变 Model 和其他的 View
  $('#input').on('change', function(){
    self.model.text = this.value;
    html = render('{{text}}', self.model);
    $('#showText').html(html);
  });

  // Controller 上的操作处理和 MVC 的方式类似
  self.event['change'] = function(){
    self.model.text = '新的 ViewA 渲染完成';
    html = render('{{text}}', self.model);
    $('#showText').html(html);
  }
}
});

```

可以看出，这时 View 和 Model 主要用于提供视图模板和数据而不做任何逻辑处理，这是有好处的，因为我们现在只要关注 Presenter 上面的逻辑操作就可以了，它的职责很清晰，Presenter 作为中间部分连接 Model 和 View 的通信交互完成所有的逻辑操作，但这样 Presenter 层的内容就可能变得很重了。另外用户在 View 上的操作会反馈到 Presenter 中进行 Model 修改，并更新其他对应部分的 View 内容。

### 4.2.3 前端MVVM模式

MVVM 则可以认为是一个自动化的 MVP 框架，并且使用 ViewModel 代替了 Presenter，即数据 Model 的调用和模板内容的渲染不需要我们主动操作，而是 ViewModel 自动来触发完成，任何用户的操作也都是通过 ViewModel 的改变来驱动的。

如图 4-4 所示，用户进行操作时，ViewModel 会捕获数据变化，直接将变化反映到 View 层上。ViewModel 的数据操作最终在页面上以 Directive 的形式体现，通过对 Directive 的识别来渲

染数据或绑定事件，管理起来更清晰。那么什么是 Directive 呢？

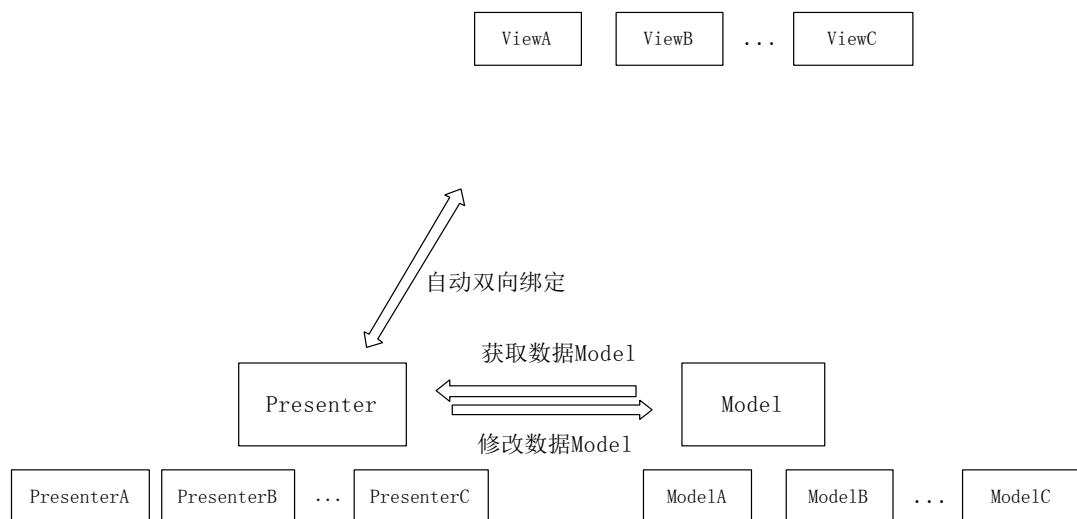


图 4-4 MVP 模式组件结构示意图

MVVM 设计的一个很大的好处是将 MVP 中 Presenter 的工作拆分成多个小的指令步骤，然后绑定到相对应的元素中，根据相对应的数据变化来驱动触发，自动管理交互操作，同时也免去了查看 Presenter 中事件列表的工作，而且一般 ViewModel 初始化时会自动进行数据绑定，并将页面中所有的同类操作复用，大大节省了我们自己进行内容渲染和事件绑定的代码量。用 MVVM 模式实现上面的例子代码如下。

```
<div id="A" q-on="click: change">
  <input type="text" q-value="text"><span q-html="text"></span>
</div>
```

```
let viewModel = new VM({
  $el: document.getElementById('A'),
  data: {
    text: 'ViewA 渲染完成'
  },
  method: {
    change() {
      this.text = '新的 ViewA 渲染完成';
    }
  }
});
```

整体上简洁了很多，模板数据的渲染和数据绑定可以通过 `q-html` 或 `q-click` 等特殊的属性来控制完成，这些特殊的元素标签属性就是我们所说的 **Directive**，当然不同的 MVVM 框架使用的 **Directive** 前缀不一样，但作用是类似的。我们再来看一个更复杂的例子。

```
<form action="#" id="form">
  <label for="text" q-html="label"></label>
  <input type="text" q-value="value" q-model="value" q-mydo="number | getValue">
  <button q-on="click: submit"></button>
</form>

let viewModel = new VM({
  $el: document.getElementById('form'),
  data: {
    label: '用户名',
    value: '输入初始值',
    number: 0
  },
  method: {
    submit() {
      // doSubmit
    }
  },
  directive: {
    mydo(value) {
      console.log(value);
    }
  },
  filter: {
    getValue() {
      return ++ value;
    }
  }
});
```

在这段代码中，**ViewModel** 初始化时自动进行了数据填充、数据双向绑定和事件绑定。我们再来看一下执行 `new VM()` 时进行 **ViewModel** 初始化所完成的事情。

如图 4-5 所示，首先 JavaScript 会找到 `document.getElementById('form')` 这个元素并开始扫描元素节点，对这个元素的属性节点 `attributes` 和所有子节点中的 `attributes` 进行遍历，一旦遍历到名称中含有 `q-` 开头的属性时，就认为是 MVVM 框架自定义的 **Directive**，此时会执行相对应的操作。例如遍历到 `q-html="label"` 时，就将 **ViewModel** 初始化时默认数据对象 `data` 中的 `label` 值赋给这个元素的 `innerHTML`；遍历到 `q-on="click: submit"` 时，就在这

个元素上绑定 click 事件, 事件函数名为 submit; 也可以自定义 q-mydo 的指令, 遍历到该节点属性时, 调用 Directive 中的 mydo 方法, 输入参数为 data 中 getValue 方法的返回值, 这里 getValue() 将输入的数字值自动加 1 并返回, getValue 函数则一般被称为过滤器。用户在 View 层操作时会自动改变 ViewModel 的数据, 然后 ViewModel 会检测数据变化, 重新遍历扫描节点属性, 执行对应的 Directive, 渲染 HTML 视图或做事件绑定。

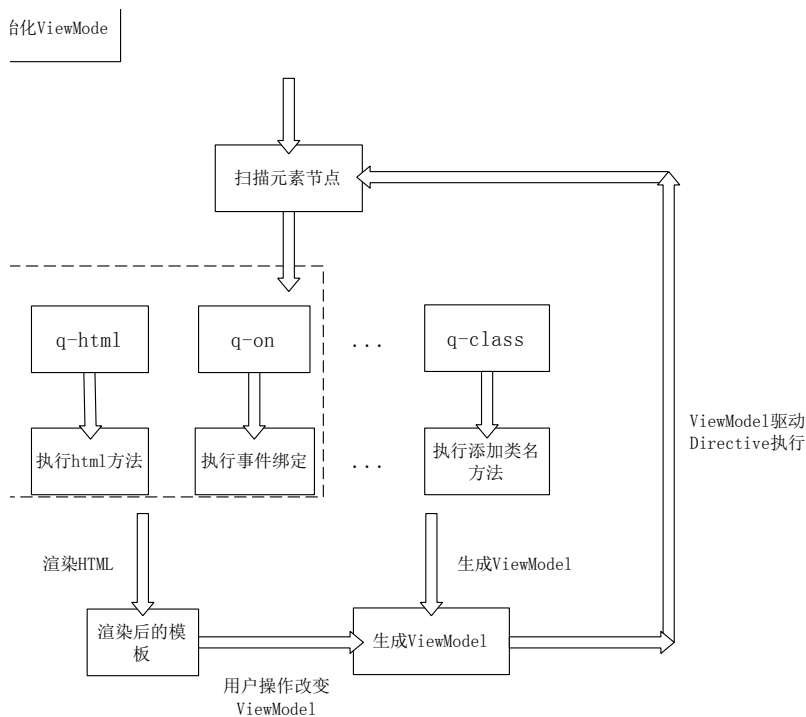


图 4-5 MVVM 初始化解析过程

这里要知道的是, q-开头的标签属性被称为指令, 这是框架约定的, 不同的框架约定的通常不一样, 例如 ng-、v-、ms-, 相信大家也见过甚至用过。这里 ViewModel 创建并进行视图渲染和事件绑定的过程非常简单, 按照这个思路去扩充, 我们就可以自己实现一个简单的 MVVM 框架了。当然完整的框架涉及的东西远比这要多, 如含有丰富的 directive、filter、表达式、ViewModel 中完善的 API, 甚至包含一些浏览器兼容性处理等。

directive、filter 具体是什么呢? 我们结合 MVVM 框架设计的相关内容来具体了解一下。

- Directive。翻译为指令, 简单地说就是自定义的执行函数, 例如 q-html、q-class、q-on、q-show、q-attr 等封装了 DOM 的一些基本可复用性的操作函数 API。



- **Filter**。也叫过滤器，如 `bool`、`upperCase`、`lowerCase` 等，指用户希望对传入的初始数据进行处理，然后再将这个处理的结果交给 **Directive** 或下一个 **Filter**。例如，**ViewModel** 初始化时传入的是一个时间戳，而我们希望在页面上显示格式化后的时间，那么就可以用 `q-html="time | formartTime"` 这样的方式来使用，其中 `formatTime` 则是将时间戳 `time` 转化为时间格式的 **Filter** 函数。
- **表达式设计**。如 `if...else` 等，类似前端普通的页面模板表达式，其作用也是控制页面内容按照具体条件来显示。
- **ViewModel 设计**。是实现传入的 **Model** 数据在内存中存放的环节，通常 **ViewModel** 也会提供一些基本的操作 **API**，方便开发者对数据进行读取或修改。
- **数据变更检测**。我们上面讲到了 **MVVM** 通常是通过数据改变来驱动的，这样就需要进行数据的双向绑定。一般若要根据 **View** 层的变化来改变 **Model**，是通过一些特殊元素（例如 `<input>`、`<select>`、`<textarea>` 等元素）的 `onchange` 事件来触发修改 **JavaScript** 中 **ViewModel** 对象数据的内容来实现的，这点比较容易理解。另一方面是 **ViewModel** 修改，如何触发 **View** 的自动更新或重渲染呢。这种根据数据的变化来自动触发其他操作的机制就是我们说的数据变更检测，实现数据变更检测的方法主要有手动触发绑定、脏数据检测、对象劫持、**Proxy** 等。下面就来具体讲解 **ViewModel** 数据变更检测的实现方案。

## 4.2.4 数据变更检测示例

之所以将数据变更检测单独作为一节来讲解是因为其实现方式较多，而且数据变更检测除了 **MVVM** 框架设计的场景，前端很多其他地方也都可以用到。**JavaScript** 发展到 **ECMAScript 6+**，形成了更多的方式来实现数据对象的变更检测，下面以目前四种可行的方法为例一一为大家介绍。

### ✎ 手动触发绑定

手动触发指令绑定是比较直接的实现方式，主要思路是通过在数据对象上定义 `get()` 方法和 `set()` 方法（当然也可以使用其他命名方法），调用时手动触发 `get()` 或 `set()` 函数来获取、修改数据，改变数据后会主动触发 `get()` 和 `set()` 函数中 **View** 层的重新渲染功能。前面提到了，根据视图 **View** 来驱动 **ViewModel** 变化的场景主要应用于 `<input>`、`<select>`、`<textarea>` 等元素中，当用户输入内容变化时，通过监听 **DOM** 的 `change`、`select`、`keyup` 等

事件来触发操作改变 `ViewModel` 的数据。我们来看一个简单的数据双向绑定的例子。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>data-binding-method-set</title>
</head>
<body>
<input q-value="value" type="text" id="input">
<span q-text="value" id="el"></span>
<script>
  let elems = [document.getElementById('el'), document.getElementById('input')];
  let data = {
    value: 'hello'
  };

  // 定义Directive
  let directive = {
    text: function(text){
      this.innerHTML = text;
    },
    value: function(value){
      this.setAttribute('value', value);
    }
  };

  // 数据绑定监听
  if(document.addEventListener){
    elems[1].addEventListener('keyup', function(e) {
      ViewModelSet('value', e.target.value);
    }, false);
  }else{
    elems[1].attachEvent('onkeyup', function(e) {
      ViewModelSet('value', e.target.value);
    }, false);
  }

  // 开始扫描节点
  scan();
  // 设置页面 2 秒后自动改变数据更新视图
  setTimeout(function(){
    ViewModelSet('value', 'hello ouvenzhang');
  },1000)

  function scan() {
    // 扫描带指令的节点属性
    for(let elem of elems){
```

```

    elem.directive = [];
    for(let attr of elem.attributes) {
        if (attr.nodeName.indexOf('q-') >= 0) {
            // 调用属性指令
            directive[attr.nodeName.slice(2)].call(elem, data[attr.nodeValue]);
            elem.directive.push(attr.nodeName.slice(2));
        }
    }
}

// 设置数据改变后扫描节点
function ViewModelSet(key, value){
    data[key] = value;
    scan();
}
</script>
</body>
<html>

```

通过浏览器加载执行这个页面结构，**ViewModel** 的变化会自动改变输入框的内容，同样，输入内容的变化也会驱动 **ViewModel** 数据的变化。我们通过 **ViewModelSet()** 方法改变 **ViewModel** 的数据后，需要主动调用 **scan()** 方法重新扫描 **HTML** 页面上的节点，并在需要的地方重新渲染 **HTML** 结构。

### 脏检测机制

以典型的 **MVVM** 框架 **Angularjs** 为例，**Angularjs** 是通过检查脏数据来进行 **View** 层操作更新的，但我们并不针对某个框架来分析脏数据检测的机制，因为成熟框架的实现考虑了很多全面的东西，比较复杂，我们需要的是通过简洁明了的代码演示脏数据检测的实现原理。脏检测的基本原理是在 **ViewModel** 对象的某个属性值发生变化时找到与这个属性值相关的所有元素，然后再比较数据变化，如果变化则进行 **Directive** 指令调用，对这个元素进行重新扫描渲染。用这种方法实现上面的例子，代码如下。

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>data-binding-drity-check</title>
</head>
<body>
    <input qg-event="value" qg-bind="value" type="text" id="input">
    <span qg-event="text" qg-bind="value" id="el"></span>
    <script>
        let elems = [document.getElementById('el'), document.getElementById('input')];
    
```

```
let data = {
  value: 'hello'
};
// 定义Directive
let directive = {
  text: function(str) {
    this.innerHTML = str;
  },
  value: function(str) {
    this.setAttribute('value', str);
  }
};

// 初始化扫描节点
scan(elems);
$digest('value');

/**
 * 输入框数据绑定监听
 */
if(document.addEventListener){
  elems[1].addEventListener('keyup', function(e) {
    data.value = e.target.value;
    $digest(e.target.getAttribute('qg-bind'));
  }, false);
}else{
  elems[1].attachEvent('onkeyup', function(e) {
    data.value = e.target.value;
    $digest(e.target.getAttribute('qg-bind'));
  }, false);
}

setTimeout(function() {
  data.value = 'hello ouvenzhang';
  // 执行$digest 方法来启动脏检测
  $digest('value');
}, 2000)

function scan(elems) {
  // 扫描带指令的节点属性
  for(let elem of elems){
    elem.directive = [];
  }
}

// 可以理解为数据劫持监听
function $digest(value){
  let list = document.querySelectorAll('[qg-bind='+ value + ']');
```

```

        digest(list);
    }

    // 脏数据循环检测
    function digest(elems) {
        // 扫描带指令的节点属性
        for (let i = 0, len = elems.length; i < len; i++) {
            let elem = elems[i];
            for (let j = 0, len1 = elem.attributes.length; j < len1; j++) {
                let attr = elem.attributes[j];
                if (attr.nodeName.indexOf('qg-event') >= 0) {
                    // 调用属性指令
                    let dataKey = elem.getAttribute('qg-bind') || undefined;
                    // 进行脏数据检测, 如果数据改变, 则重新执行指令, 否则跳过
                    if (elem.directive[attr.nodeValue] !== data[dataKey]) {
                        directive[attr.nodeValue].call(elem, data[dataKey]);
                        elem.directive[attr.nodeValue] = data[dataKey];
                    }
                }
            }
        }
    }
}
</script>
</body>
</html>

```

其实这里的和手动绑定扫描节点的方式类似, 不同的是, 脏检测只针对可能修改的元素进行扫描, 这样就提高了 ViewModel 内容变化后扫描视图渲染的效率。

### 👉 前端数据对象劫持(Hijacking)

数据劫持是目前使用比较广泛的方式。其基本思路是使用 `Object.defineProperty` 和 `Object.defineProperties` 对 ViewModel 数据对象进行属性 `get()` 和 `set()` 的监听, 当有数据读取和赋值操作时则扫描元素节点, 运行指定对应节点的 Directive 指令, 这样 ViewModel 使用通用的等号赋值就可以了。具体例子如下。

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>data-binding-hijacking</title>
</head>
<body>
    <input q-value="value" type="text" id="input">
    <div q-text="value" id="el"></div>
    <script>

```

```
let elems = [document.getElementById('el'), document.getElementById('input')];
let data = {
  value: 'hello'
};
// 定义Directive
let directive = {
  text: function(text) {
    this.innerHTML = text;
  },
  value: function(value) {
    this.setAttribute('value', value);
  }
};

let bValue;
scan();

// 可以理解为数据劫持监听
defineGetAndSet(data, 'value');

// 数据绑定监听
if(document.addEventListener){
  elems[1].addEventListener('keyup', function(e) {
    data.value = e.target.value;
  }, false);
}else{
  elems[1].attachEvent('onkeyup', function(e) {
    data.value = e.target.value;
  }, false);
}

setTimeout(function() {
  data.value = 'hello ouvenzhang';
}, 2000);

function scan() {
  // 扫描带指令的节点属性
  for(let elem of elems){
    elem.directive = [];
    for(let attr of elem.attributes) {
      if (attr.nodeName.indexOf('q-') >= 0) {
        // 调用属性指令
        directive[attr.nodeName.slice(2)].call(elem, data[attr.nodeValue]);
        elem.directive.push(attr.nodeName.slice(2));
      }
    }
  }
}
```

```
// 定义对象属性设置劫持
function defineGetAndSet(obj, propName) {
  Object.defineProperty(obj, propName, {
    get: function() {
      return bValue;
    },
    set: function(newValue) {
      bValue = newValue;
      scan();
    },
    enumerable: true,
    configurable: true
  });
}
</script>
</body>
</html>
```

需要注意的是，`defineProperty` 只支持 Internet Explorer 8 以上和 Chrome 等标准的浏览器，且 Internet Explorer 8 浏览器中需要使用 `es5-shim` 来提供支持。Firefox 浏览器不支持该方法，需要使用 `__defineGetter__` 和 `__defineSetter__` 来代替，这里为了让大家理解其中的原理，所以直接使用了 `defineProperty` 来实现。

### 👉 ECMAScript 6 Proxy

在前面章节中，我们讲解 ECMAScript 6 时向大家介绍了 `Proxy` 特性，它可以用于在已有的对象基础上重新定义一个对象，并重新定义对象原型上的方法，包括 `get()` 方法和 `set()` 方法，同时我们也将它和 `defineProperty` 进行了比较。下面来看一下使用 `Proxy` 如何进行数据的变更检测。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>data-binding-proxy</title>
</head>

<body>
  <input q-value="value" type="text" id="input">
  <div q-text="value" id="el"></div>
  <script>

    'use strict';
    let elems = [document.getElementById('el'), document.getElementById('input')];
    // 定义 Directive
```

```
let directive = {
  text: function(text) {
    this.innerHTML = text;
  },
  value: function(value) {
    this.setAttribute('value', value);
  }
};

// 设置 data 的访问 Proxy
let data = new Proxy({}, {
  get: function (target, key, receiver) {
    return target.value;
  },
  set: function (target, key, value, receiver) {
    target.value = value;
    scan();
    return target.value;
  }
});

data['value'] = 'hello';
scan();
/**
 * 数据绑定监听
 */
if(document.addEventListener){
  elems[1].addEventListener('keyup', function(e) {
    data.value = e.target.value;
  }, false);
}else{
  elems[1].attachEvent('keyup', function(e) {
    data.value = e.target.value;
  }, false);
}

setTimeout(function() {
  data.value = 'hello ouvenzhang';
}, 2000);

function scan() {
  // 扫描带指令的节点属性
  for(let elem of elems){
    elem.directive = [];
    for(let attr of elem.attributes) {
      if (attr.nodeName.indexOf('q-') >= 0) {
        // 调用属性指令
        directive[attr.nodeName.slice(2)].call(elem, data[attr.nodeValue]);
      }
    }
  }
}
```



```

        elem.directive.push(attr.nodeName.slice(2));
    }
}
}
}
</script>
</body>
</html>

```

这里通过简单的代码体现了 MVVM 双向数据绑定的基本原理，希望读者们认真阅读并理解。关于 MVVM 的数据变更检测介绍的比较多，因为可以认为这是 MVVM 设计实现的最关键部分，如果处理得好会大大提升 MVVM 框架的效率，否则会有较多重复的元素扫描过程而拖累代码执行速度。到这里相信大家也对 MVVM 的设计原理有了较深的了解，自己也可以实现一个类似的框架。

总结来看，前端框架从直接 DOM 操作到 MVC 设计模式，然后到 MVP，再到 MVVM 框架，前端设计模式的改进原则一直向着高效、易实现、易维护、易扩展的基本方向发展。虽然目前前端各类框架也已经成熟并开始向高版本迭代，但是还没有结束，我们现在的编程对象依然没有脱离 DOM 编程的基本套路，一次次框架的改进大大提高了开发效率，但是 DOM 元素运行的效率仍然没有变。要解决这个问题，就必须了解下一节中介绍的前端 Virtual DOM。

## 4.3 Virtual DOM交互模式

### 4.3.1 Virtual DOM设计理念

MVVM 的前端交互模式大大提高了编程效率，自动双向数据绑定让我们可以将页面逻辑实现的核心转移到数据层的修改操作上，而不再是在页面中直接操作 DOM。但实际上，通过上一节的内容可以看出，尽管 MVVM 改变了前端开发的逻辑方式，但是最终数据层反应到页面上 View 层的渲染和改变仍是通过对应的指令进行 DOM 操作来完成的，而且通常一次 ViewModel 的变化可能会触发页面上多个指令操作 DOM 的变化，带来大量的页面结构层 DOM 操作或渲染。先来看下面这个应用场景。

```

<ul id="root">
  <li q-repeat="list">
    <span q-text="value"></span>
    <span>固定文本</span>
  </li>
</ul>

```

```
let viewModel = new VM({
  $el: document.getElementById('root'),
  data: {
    list: [{value: 1}, {value: 2}, {value: 3}]
  }
});
```

使用 MVVM 框架时就生成了一个数字列表，此时如果需要显示的内容变成了 [{value: 0}, {value: 1}, {value: 2}, {value: 3}]，在 MVVM 框架中一般会重新渲染整个列表，包括列表中无须改变的部分也会重新渲染一次。但实际上如果直接操作改变 DOM 的话，只需要在<ul>子元素前插入一个新的<li>元素就可以了。但在一般的 MVVM 框架中，我们通常不会这样做。毫无疑问，这种情况下 MVVM 的 View 层更新模式就消耗了更多没必要的性能。

那么该如何对 ViewModel 进行改进，让浏览器知道实际上只是增加了一个元素呢？通过对比 [{value: 1}, {value: 2}, {value: 3}] 和 [{value: 0}, {value: 1}, {value: 2}, {value: 3}]，我们发现其实只是增加了一个 {value: 0}，那么该怎样将这个增加的数据反映到 View 层上呢？我们可以这样想，将新的 Model data 和旧的 Model data 进行对比，然后记录 ViewModel 的改变方式和位置，就知道了这次 View 层应该怎样去更新，这样比直接重新渲染整个列表高效得多。

这里其实可以理解为，ViewModel 里的数据就是描述页面 View 内容的另一种数据结构标识，不过需要结合特定的 MVVM 描述语法编译来生成完整的 DOM 结构。试想一下，如果 ViewModel 里的数据和 MVVM 的语法结构放在一起用另一种对象表示，代码可能如下。

```
let ulElement = {
  tagName: 'ul',
  attributes: [{
    id: 'root'
  }],
  children: [
    {tagName: 'li', children: [{
      tagName: 'span',
      nodeText: 1
    }], {
      tagName: 'span',
      nodeText: '固定文本'
    }},
    {tagName: 'li', children: [{
      tagName: 'span',
      nodeText: 2
    }], {
      tagName: 'span',
```

```

        nodeText: '固定文本'
      }],
      {tagName: 'li', children: [{
        tagName: 'span',
        nodeText: 3
      }, {
        tagName: 'span',
        nodeText: '固定文本'
      }]}
    ]
  };

```

如图 4-6，如果用 JavaScript 对象的属性层级结构来描述上面 HTML DOM 对象树的结构，就可以这样来表示。当数据改变时，新生成一份改变后的 `ulElement`，并与原来的 `ulElement` 结构进行对比，可以看出，上面的操作只需要在 `ulElement` 对象 `children` 属性的最前面增加以下内容即可。

```

{tagName: 'li', children: [{
  tagName: 'span',
  nodeText: 0
}], {
  tagName: 'span',
  nodeText: '固定文本'
}}

```

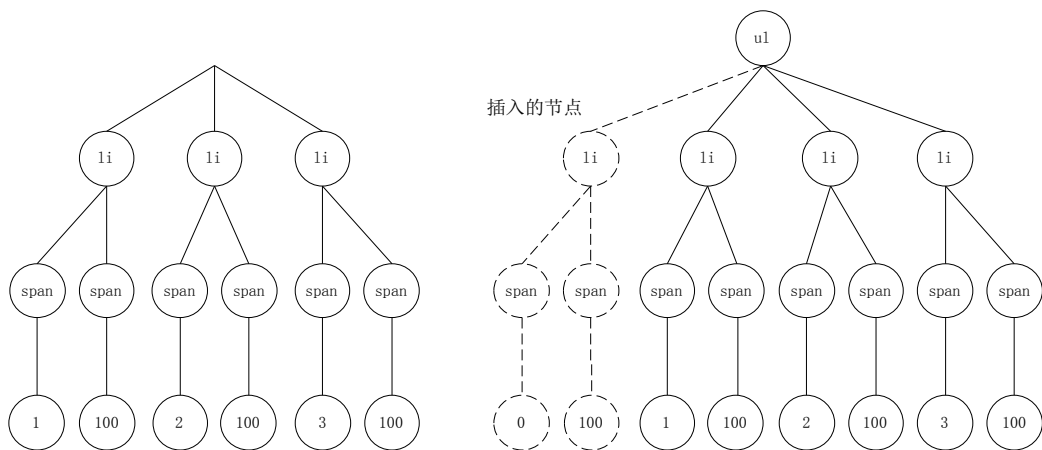


图 4-6 节点变化对比

这样我们就找到了进行这次视图改变的最小操作描述的对象，根据这个差异性描述的对象，可以很容易地创建出变更的 DOM 结构，完成这次 HTML DOM 结构的修改，而不是直接对整个列表重新渲染。这样我们也不用再去读取 MVVM 的语法结构了，否则仍需扫描 DOM 节点的

所有属性,进行较多的 DOM 操作,而浏览器中 DOM 对象的默认属性是很多的,这样 ViewModel 初始化渲染的过程便会很慢。

这里的 ulElement 对象可以理解为 Virtual DOM。通常认为, Virtual DOM 是一个能够直接描述一段 HTML DOM 结构的 JavaScript 对象,浏览器可以根据它的结构按照一定规则创建出确定唯一的 HTML DOM 结构。整体来看, Virtual DOM 的交互模式减少了 MVVM 或其他框架中对 DOM 的扫描或操作次数,并且在数据发生改变后只在合适的地方根据 JavaScript 对象来进行最小化的页面 DOM 操作,避免大量重新渲染。

### 4.3.2 Virtual DOM的核心实现

先总结一下使用 Virtual DOM 模式来控制页面 DOM 结构更新的过程:创建原始页面或组件的 Virtual DOM 结构,用户操作后需要进行 DOM 更新时,生成用户操作后页面或组件的 Virtual DOM 结构并与之前的结构进行对比,找到最小变化 Virtual DOM 的差异化描述对象,最后把差异化的 Virtual DOM 根据特定的规则渲染到页面上。所以核心操作可以抽象成三个步骤:创建 Virtual DOM;对比两个 Virtual DOM 生成差异化 Virtual DOM;将差异化 Virtual DOM 渲染到页面上。下面逐一了解这三个步骤的具体操作。

#### 👉 创建Virtual DOM

创建 Virtual DOM 即把一段 HTML 字符串文本解析成一个能够描述它的 JavaScript 对象。我们很自然地想,通过浏览器提供的 DOM API 扫描这段 DOM 的节点,遍历它的属性,然后添加到 JavaScript 对象上去即可。

```
<ul id="root">
  <li>
    <span>1</span>
    <span>固定文本</span>
  </li>
  <li>
    <span>2</span>
    <span>固定文本</span>
  </li>
  <li>
    <span>3</span>
    <span>固定文本</span>
  </li>
</ul>
```

例如上面这段 HTML 片段,遍历完成后生成如下结构。

```

let ulElement = {
  tagName: 'ul',
  attribute: [{
    id: 'root'
  }],
  children: [
    {tagName: 'li', children: [{
      tagName: 'span',
      nodeText: 1
    }], {
      tagName: 'span',
      nodeText: '固定文本'
    }]},
    {tagName: 'li', children: [{
      tagName: 'span',
      nodeText: 2
    }], {
      tagName: 'span',
      nodeText: '固定文本'
    }]},
    {tagName: 'li', children: [{
      tagName: 'span',
      nodeText: 3
    }], {
      tagName: 'span',
      nodeText: '固定文本'
    }]}
  ]
};

```

这似乎很合理，但其实这样是错的。这样创建 Virtual DOM 会直接失去 Virtual DOM 的优势，它是为了避免直接进行 DOM 操作而设计的。我们不能通过浏览器 DOM API 扫描去生成 JavaScript 对象，因为扫描过程本身使用到 DOM 的读取操作，这个过程很慢。一种可选的方式是，直接书写类似 ulElement 的 JavaScript 结构表示 Virtual DOM，但这样又会导致层次不清晰，当节点过多时，JavaScript 对象就会很大，难以阅读和开发管理。相比之下我们仍然习惯使用 HTML 标记来书写最终浏览器页面的结构，所以一种更可选的方法是，自己实现上述这段 HTML 字符串文本的解析方式，根据标签之间的关系，读取生成 Virtual DOM 的结构。例如提供某个常见 Virtual DOM 的方法 createVDOM。

```

let htmlString = `


```

```

    <span>2</span>
    <span>固定文本</span>
  </li>
</li>
  <span>3</span>
  <span>固定文本</span>
</li>
</ul>;`

```

```
let ulElement = createVDOM(htmlString);
```

那么 `createVDOM` 就可以如下实现：逐个分析字符串中的字符，根据词法分析内容，将标签名字存为 `tagName`，属性存入 `attributes`，子标签内容存入 `children`。通过这种方式，我们可以将一段 HTML 文本字符串解析成一个 JavaScript 对象。到目前为止，浏览器对 HTML 还没有任何处理，而是 JavaScript 直接分析 HTML 字符串文本来生成 Virtual DOM，所以这就比 DOM API 操作要快。创建 Virtual DOM 往往就是将一段 DOM 描述字符串解析成 Virtual DOM 对象的过程，供后面操作调用。

图 4-7 为浏览器进行词法分析的状态转换图。根据 HTML 字符串解析创建 Virtual DOM 的过程相当于实现了一个 HTML 文本解析器，但是没有生成 DOM 对象树，只是生成了一个操作效率更高的 JavaScript 对象，因此通常不会直接将 HTML 交给浏览器去解析，因为浏览器的 DOM 解析很慢，这也是 Virtual DOM 交互模式和普通 DOM 编程最本质的区别。完成之后再通过 Virtual DOM 进行渲染生成一个真实的 DOM 操作就比较简单了。

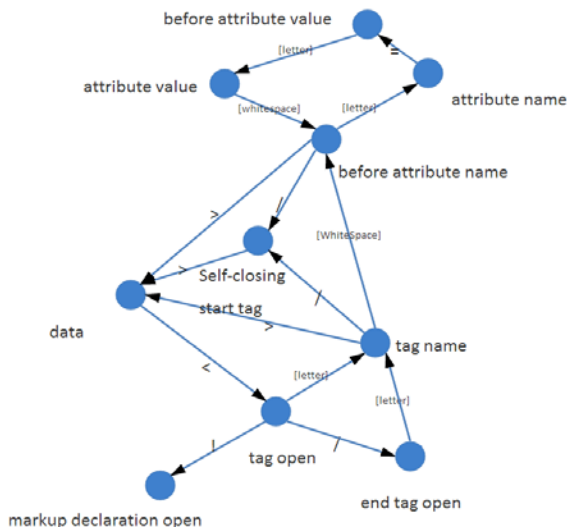


图 4-7 文法解析过程 HTML

```

const render = function (virtualDOM) {
  let element = document.createElement(virtualDOM.tagName);
  let attributes = virtualDOM.attributes;

  // 设置节点的 DOM 属性
  for (let key in attributes) {
    element.setAttribute(key, attributes[key])
  }

  let children = virtualDOM.children || []

  for(let child of children){
    // 如果是字符串则直接插入字符串，否则构建子节点
    let childNode = (typeof child === 'string') ? document.createTextNode(child) :
render(child)
    element.appendChild(childNode)
  }

  return element;
}

```

### 👉 对比 Virtual DOM

当用户进行了页面操作需要进行页面视图改变时，通常会生成一个新的 Virtual DOM 结构来表示改变后的状态，而且不会将这个改变后的 Virtual DOM 内容立即重新渲染到页面中，而是通过对比找出两个 Virtual DOM 的差异性，得到一个差异树对象。对于 Virtual DOM 的对比算法实际上是对多叉树结构的遍历算法。对多叉树遍历就有广度优先算法和深度优先算法，我们以深度优先算法为例来看一下 Virtual DOM 树的对比过程。

如图 4-8 所示，可以对 Virtual DOM 中的每个节点添加一个唯一的字母 id，那么两个 Virtual DOM 的节点顺序分别使用深度优先遍历算法表示为 ABEFCGHDIJ 和 AKLMBEFCGHDIJ，这样我们很容易分析出需要在 A 和 B 之间进行插入操作 KLM 节点，再根据 KLM 的关系，可以知道只需要插入完整的 K 节点即可。使用广度优先算法遍历的思路也是类似的，遍历出两个 Virtual DOM 节点顺序为 ABCDEFGHIJ 和 AKBCDLMEFGHIJ，不过稍微不同的是，这种情况下检测到有两处插入，需要进一步判断来合并操作，优化差异树的结构。

此外在 Virtual DOM 的对比过程中，除了节点改变的内容，还需要继续记录发生差异化改变的类型和位置，例如是针对具体哪一个元素的增加、替换、删除操作等。

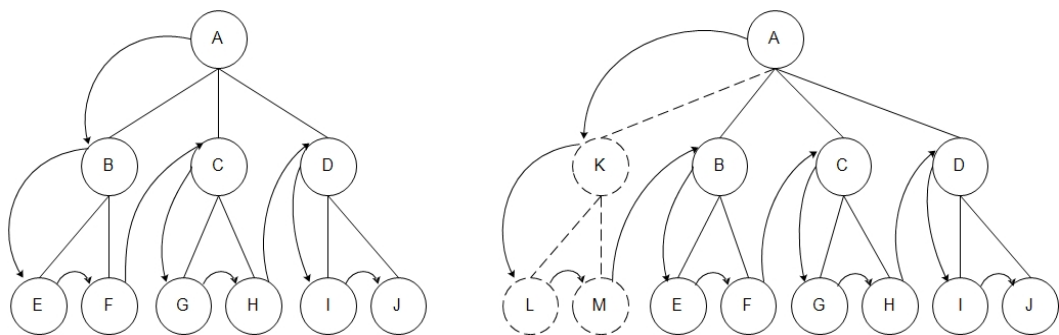


图 4-8 深度优先算法对比 Virtual DOM

### 渲染差异化 Virtual DOM

经过 Virtual DOM 的差异化对比，我们获取了用户操作后的差异化 Virtual DOM、差异化类型和差异化的位置，那么剩下的操作就很直接了，根据对比返回的结果将差异化内容经过 DOM 操作渲染到页面上，整个交互的过程便完成了。

与以前交互模式相比，Virtual DOM 最本质的区别在于减少了对 DOM 对象的操作，通过 JavaScript 对象来代替 DOM 对象树，并且在页面结构改变时进行最小代价的 DOM 渲染操作，提高了交互的性能和效率。这就是 Virtual DOM 交互模式的优势，也是提高前端交互性能的根本原因。

参考资料：<http://w3c.github.io/html/syntax.html>。

通过 Virtual DOM 的实现分析，我们大概了解了它的实现过程和主要优势。这种直接解析和操作 JavaScript 对象的方式相对于频繁的 DOM 操作速度更快，可以认为 Virtual DOM 的设计是为了提升页面的渲染性能，为前端构建高性能的 Web 应用提供了新的实践方案，在实践项目中非常值得尝试和推广。但实际上，我们只是通过这种方式减少了 DOM 操作，而没有完全脱离 DOM 操作。在最后的渲染阶段，DOM 的渲染操作仍然无法避免，在浏览器 DOM 性能更慢的移动端 Hybrid WebView 中，如果想完全脱离对 DOM 的操作来进一步提高性能，又该怎么做呢？

## 4.4 前端MNV\*时代

尽管 Virtual DOM 的交互模式能在页面数据渲染和变更时尽可能地减少 DOM 操作，但仍无法完全脱离 DOM 交互的模式。我们知道 DOM 的操作效率不高，在移动设备的 Hybrid



WebView 上表现会更慢, 所以为了进一步改进 Hybrid 应用中的 DOM 性能, 希望完全脱离 DOM 编程的模式来进行结构层的操作。

为什么可以这样来实现呢? 首先, 目前主流 Hybrid App 的 Web 内容通常是在原生应用中嵌入 WebView 来实现的, 而原生应用的界面数据渲染可以通过调用原生控件来实现, 它不仅没有 HTML DOM 的性能缺陷, 而且还可以直接调用 Native 系统底层的 API; 其次, 我们在第 2 章中讲到, Hybrid App 可以通过统一的 JavaScript 交互协议来调用原生的方法和控件。所以使用 JavaScript 直接调用和产生原生控件进行界面数据渲染的方式是可以实现的。

#### 4.4.1 MNV\*模式简介

我们把这种使用 JavaScript 调用原生控件或事件绑定来生成应用程序的交互模式称为前端 MNV\*开发模式。可以简单理解为 Model-NativeView-\*, 而后面的\*可以表示 Virtual DOM 或 MVVM 中的 ViewModel, 我们也可以自己使用 Controller 来实现调用的方式。这样定义是非常合适的, 相比之前的不同无非就是使用 NativeView 代替了 View。

如果说 Virtual DOM 减少了 DOM 的交互次数, 那么 MNV\*想要做的一件事情就是完全抛弃使用 DOM, 而交互数据的操作依然可以使用 ViewModel、Virtual DOM 或者直接的 Model 来实现, 具体就看实现的方式了。值得注意的是, 这种模式目前仅适用于移动端 Hybrid 应用, 因为需要依赖原生应用控件的调用支持, 而只有这种特殊的应用场景才满足条件。

#### 4.4.2 MNV\*模式实现原理

我们直接来看一下 MNV\*模式的通用实现思路。显示一段文本内容代码如下。

```
<TextView q-text="text"></TextView>
```

对应的数据 Model 为 {text: 'hello'}, 解析到 Native 端生成一个文本域的过程就可以这样来描述。

如图 4-9 所示, 以 ViewModel 封装实现生成 NativeView 的思路为例, 我们需要通过 NativeView 来渲染一段文本, 需要注意的是, 此时开发的前端结构层规范就不是 HTML 了, 而是 XML 规范, 两者解析过程大致类似。首先, MNV\*框架需要解析数据 ViewModel 和 XML 的指令 Directive, 我们知道 JavaScript 可以借助通用协议来调用原生应用的方法, 然后按照标准的 JSBridge 协议将 Model 和 Directive 封装成协议串 jsbridge://DOMRender:success/createView?{"text":"hello","element":"TextView"} 的形式, 通过 Prompt(Android

上的方法, iOS 上使用 `iframe`) 发送到客户端, 这里调用的是解析 `DOMRender` 的 `createView` 方法创建一个 `TextView`, `TextView` 是 Android 原生系统上的一个内置文本控件的基类, 和 HTML 的 `<p>` 标签类似, 可以用来创建移动端上的一个文本域展示一段文字。此时客户端会解析到这段协议串, 调用原生应用动态创建文本控件 `TextView` 的界面内容碎片 (类似于 HTML 的文档碎片 `documentFragment` 或一段 XML 片段), 这里 Android 中 `TextView` 的控件内容描述通过 XML 规范就可以如下表示。

```
<TextView
style="@style/text_view_style"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:layout_weight="1"
android:text="hello" />
```

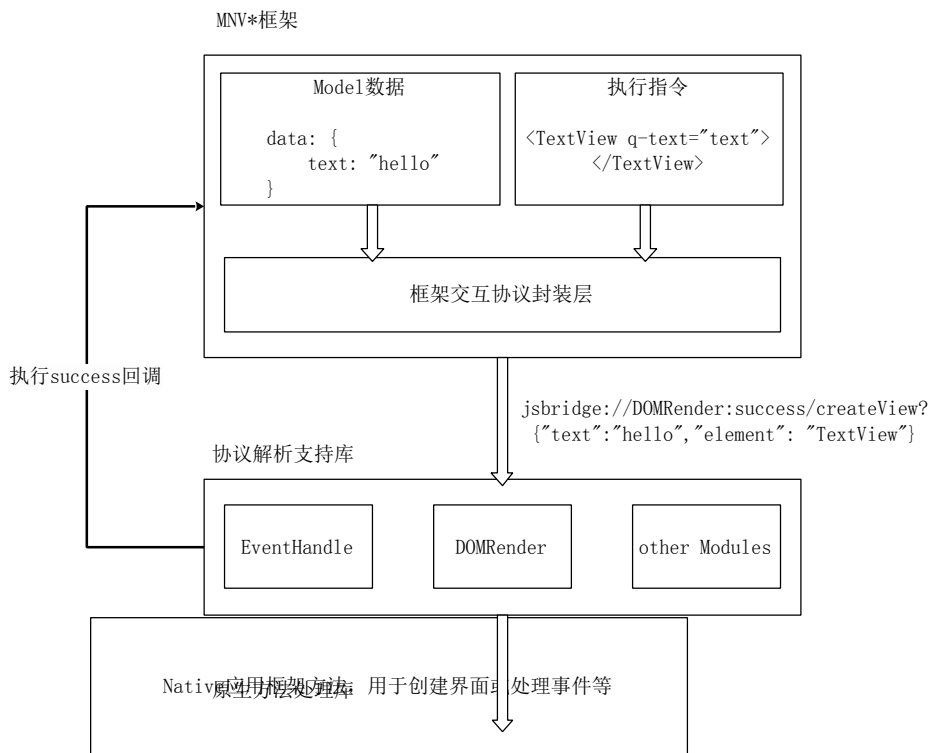


图 4-9 MNV\*开发模式设计思路

这里也可以传入基本的样式为创建成功的 `TextView` 添加一些基础样式。同样执行成功后也能回调 `JavaScript` 传入的 `success()` 方法来异步执行 `JavaScript` 的其他操作。

整体上设计实现这样一个 `Native` 渲染机制思路并不难,但是要实现好 `JavaScript` 端和 `Native` 端的封装,难度就比较大了。`MNV*` 框架端的主要任务是解析 `Model`、`ViewModel` 或 `Virtual DOM` 组成 `JSBridge` 协议串并发送,而 `Native` 端的实现将会比较复杂,需要处理不同的标签元素解析,例如遇到 `<TextView>` 标签则创建 `TextView` 控件,遇到 `<Layout>` 标签创建 `Layout` 控件,还可能需处理事件的绑定等,即将 `JavaScript` 的事件通过 `Native` 事件来实现。整体上像是使用移动端原生的方式来解析 `HTML` 上需要实现的应用功能。

`MNV*` 的基本原理主要是将 `JSBridge` 和 `DOM` 编程的方式进行结合,让前端能够快速构建开发原生界面的应用,从而脱离 `DOM` 的交互模式。这一节简要介绍了 `MNV*` 的基本实现思路,大家也可以根据兴趣和业务场景来选择使用目前主流的 `MNV*` 开发框架进行尝试,而且目前 `MNV*` 开发模式也正处于一个推广应用阶段,相信未来会更加普及。

## 4.5 本章小结

在这一章中,我们就前端各类框架进行了原理性分析,从直接 `DOM` 编程到 `MV*` 交互模式,再到 `Virtual DOM` 编程理念,以及 `MNV*` 的渲染方式,前端框架一直秉承着提高效率和性能的宗旨一步步变化。在学习这章的同时,我们也需要深入理解体会各类框架的实现设计方式,这样才能理解前端框架的变化规律。作为前端工程师,我们需要这种追根究底的精神。下一章将开始分析讲解大型前端项目中的应用实践技术,让读者们学习和了解在实际的项目开发中应该具备哪些方面的知识和能力。

# 第5章

## 前端项目与技术实践

现代前端技术飞速发展，最终形成了以效率和质量为核心的两大趋势。就效率而言，在大型前端项目的开发中，规范的制定、框架的出现与升级、构建的使用更新、组件化的设计实现等都在于让前端能更快、更高效地完成更多的事情。质量方面，前端优化的提出、前端用户数据的收集、错误日志的收集上报等，都是为了帮助开发者来提高前端性能，提升用户体验。目前，前端已经进入了以效率和质量为核心的工业化时代，各类辅助工具和技术的使用大大减少了前端开发的重复工作量，省去了很多低效的操作。在这一章中，我们将以不同专题的形式一起来看一看前端的高效率技术和质量提高手段是如何在大型项目中实践运用的，这也是我们作为一名前端工程师必须具备的工程思维和能力。

### 5.1 前端开发规范

开发规范可以认为是软件开发工程师之间交流的另一种语言，它在一定程度上决定了团队协作过程中开发的程序代码是否具有一致性和易维护性，统一的开发规范常常可以降低代码的出错概率和团队开发的协作成本。开发规范制定的重要性不言而喻，使用怎样的规范又成为了另一个问题，因为编程规范并不唯一。通俗地讲，规范的差别很多时候只是代码写法的区别，不同的规范都有各自的特点，没有优劣之分，在选择时也没必要纠结于使用哪一种规范，不过既然规范是提高一个团队开发效率的虚拟工具，那么在一个团队里还是尽可能使用同一种开发规范比较好。

实际上，我们平时所说的开发规范更多时候指的是狭义上的编码规范，广义上的开发规范包括实际项目开发中可能涉及的所有规范，如项目技术选型规范、组件规范、接口规范、模块化规范等。由于每个团队使用的项目技术实现不一样，项目技术选型规范、组件规范、接口规

范、模块化规范等也可能千差万别，但无论是哪一种规范，推荐在一个团队中尽可能保持统一。本节中，我们先来讨论前端开发规范，主要指的是狭义上的前端编码规范。

我们将从前端通用规范、HTML 规范、CSS 规范、ECMAScript 5 规范、ECMAScript 6+规范和防御性编程等几个方面来向大家介绍前端所涉及的开发编码规范，同时也会尽量向大家介绍这样制定编码规范的原因和好处，如果有读者觉得对规范的使用都比较了解了，建议跳过这一节，继续学习下一节内容。对于本节推荐的规范，大家可以选择性借鉴，不一定作为唯一的标准学习使用。

### 5.1.1 前端通用规范

#### 📁 三层结构分离

前端页面开发应做到结构层（HTML）、表现层（CSS）、行为层（JavaScript）分离，保证它们之间的最小耦合，这对前期开发和后期维护都是至关重要的。移动端开发可以适当地进行 CSS 样式、图片资源、JavaScript 内联，内联的资源大小标准一般为 2KB 以内，否则可能会导致 HTML 文件过大，页面首次加载时间过长。

<!-- 不推荐 -->

```
<button style="background-color: #ccc;" onclick="javascript: console.log(this);">按钮</button>
```

<!-- 推荐，相关样式和 JavaScript 逻辑写在外部引入的 CSS 和 JavaScript 文件中 -->

```
<link rel="stylesheet" href="./base.css" >
...
<button class="btn btn-primary">按钮</button>
...
<script src="./base.js"></script>
```

#### 📁 缩进

统一使用 tab（或 4 个空格宽度）来进行缩进，可以在开发编辑器或 IDE 里进行设置。虽然推荐使用 4 个空格来缩进，但其实选择哪种缩进方式并不重要，重要的是在一个项目中缩进方式要保持一致，不要出现混用的情况，否则会造成阅读上的障碍。幸运的是，现在开发工具的格式化插件能帮助我们完成这件事情，所以要尽量在开发工具中设置来让工具自动完成这件事情。

#### 📁 内容编码

在 HTML 文档中用<meta charset="utf-8">来指定编码，以避免出现页面乱码问题。

不需要为 CSS 显式定义编码，其默认为 utf-8。

```
/* 不推荐 */
@charset "utf-8";

html, body{
    margin: 0;
    padding: 0;
}

/* 推荐 */
html, body{
    margin: 0;
    padding: 0;
}
```

### 👉 小写

所有的 HTML 标签、HTML 标签属性、样式名及规则建议使用小写，我们一般习惯使用小写英文字符，大写单词相对不容易阅读和理解。HTML 属性的 id 属性可以使用驼峰大小写组合的命名方式，因为 id 属性常常只用于 JavaScript 的 DOM 查询引用，而 JavaScript 语言标准推荐使用驼峰大小写组合的命名方式，因此 HTML 页面上的 id 属性也尽量使用这种标准来写。

```
<!-- 不推荐 -->
<UL id="menu_list" class="menu-list">
    <LI class="menu-list-item"></LI>
    <LI class="menu-list-item"></LI>
    <LI class="menu-list-item"></LI>
</UL>

<!-- 推荐 -->
<ul id="menuList" class="menu-list">
    <li class="menu-list-item">1</li>
    <li class="menu-list-item">2</li>
    <li class="menu-list-item">3</li>
</ul>
```

### 👉 代码单行长度限制

代码单行长度不要超过 120 字符（或 80 字符，具体可根据团队习惯来决定），长字符串拼接通常使用加号来连接换行的内容。

### 👉 注释

尽可能地为代码写上注释，无论是 HTML、CSS 还是 JavaScript，必要的注释是不能少的。

段内容描述可以使用段注释，单行内容则使用单行注释，对于独立的文件而言，也尽量在文件头部添加文件注释。当然，更推荐使用自文档化风格的代码进行开发，通过代码的含义来代替注释。

```
/**
 * filename: util.js
 * author: ouvenzhang
 * description: 提供常见的的工具函数集，主要包含
 *   getDay(): 获取中文星期时间格式，例如星期一
 *   formatTime(): 获取格式化后的中文时间表示，例如 2016 年 12 月 12 日
 *   ...
 */

let util = {};

/**
 * 获取带中文的星期字符串
 * @param {[timestamp]} timestamp [输入的时间戳]
 * @return {[string]} [返回中文星期时间表示]
 */
function _getDay(timestamp) {
  // 默认的星期表示字符串
  const Day = ['星期日', '星期一', '星期二', '星期三', '星期四', '星期五', '星期六'];
  return Day[timestamp.getDay()];
}

...

module.exports = {
  getDay: _getDay,
  ...
}
```

自文档化开发是目前比较提倡的一种书写带有具体含义项目代码的编码方式，它提出要尽可能让代码本身来表示代码执行的功能描述，而减少文档注释的书写，因为文档注释需要更多的时间去维护。在下面的例子中，第二种方式就比第一种方式更加清晰，即使不使用注释也能很容易理解代码的含义，而第一种方式一定要添加注释说明才能理解其中的含义。

```
// 代码块一
if(!el.offsetWidth && !el.offsetHeight) {}

// 代码块二
function isVisible(el) {
  return !el.offsetWidth && !el.offsetHeight;
}
```

```
if(!isVisible(el)) {}
```

再如:

```
// 代码块一
let width = (value - 0.5) * 16;

// 代码块二
let width = emToPixels(value);

function emToPixels(ems) {
  return (ems - 0.5) * 16;
}
```

### 👉 行尾空格与符号

删除行尾空格与多余的符号, 这些内容是没有必要存在的。

## 5.1.2 前端HTML规范

### 👉 文档类型定义

统一使用 HTML5 的标准文档类型<!DOCTYPE html>来定义, 这样更简洁, 而且向后兼容。不使用 HTML 4.01 的 DTD 定义。

```
<!-- 不推荐 -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/
DTD/xhtml11.dtd">

<!-- 推荐 -->
<!DOCTYPE html>
```

### 👉 head内容

head 中必须定义 title、keyword、description, 保证基本的 SEO 页面关键字和内容描述。移动端页面 head 要添加 viewport 控制页面不缩放, 有利于提高页面渲染性能。建议在页面<head>上加上基本的社交 RICH 化消息, 保证网页地址分享到主流社交平台后显示页面的缩略图、标题和描述等。

```
<!-- 推荐 -->
<meta name="viewport" content="width=device-width,minimum-scale=1.0,maximum-scale=1.0,
user-scalable=no"/>
<meta itemprop="name" content="页面标题">
<meta name="description" itemprop="description" content="页面内容描述.">
<meta itemprop="image" content="http://www.domain.com/assets/logo.png">
```



### 👉 省略type属性

在引用 CSS 或 JavaScript 时,可以省略 type 属性不写,因为 HTML5 在引入 CSS 时默认 type 值为 text/css,在引入 JavaScript 时默认 type 值为 text/javascript。

```
<!-- 不推荐 -->
<link type="text/css" rel="stylesheet" href="./base.css">
<script type="text/javascript" src="./base.js"></script>

<!-- 推荐 -->
<link rel="stylesheet" href="./base.css">
<script src="./base.js"></script>
```

### 👉 使用双引号包裹属性值

所有的标签属性值必须要用双引号包裹,不允许有的用双引号有的用单引号,这样有利于区分标签的属性名和属性值。

```
<!-- 不推荐 -->
<div class='ui-dialog'></div>

<!-- 推荐 -->
<div class="ui-dialog"></div>
```

### 👉 属性值省略

非必需的属性值可以省略。例如输入框里的 readonly、disabled 和 required 等属性值是非必需的,可以省略不写。

```
<!-- 不推荐 -->
<input type="text" readonly="readonly">
<input type="text" disabled="disabled">

<!-- 推荐 -->
<input type="text" readonly>
<input type="text" disabled>
```

### 👉 嵌套

所有元素必须正确嵌套,尽量使用语义化标签,不允许交叉,也不允许在 inline 元素中包含 block 元素。

```
<!-- 不推荐 -->
<span>
  <div>这是一个块级 div 元素<p>
    </div>这是一个块级 p 元素</p>
</span>
```

```
<ul>
  <h3>list</h3>
  <li>ouven</li>
  <li>zhang</li>
</ul>

<!-- 推荐 -->
<div>
  <p>这是一个块级 div 元素</p>
  <p>这是一个块级 p 元素</p>
</div>
<div>
  <h3>list</h3>
  <ul>
    <li>ouven</li>
    <li>zhang</li>
  </ul>
</div>
```

### 👉 标签闭合

非自闭合标签必须添加关闭标识，自闭合标签无须关闭。在 W3C 的不同规范中，标签的闭合检查也是不一样的。XHTML 较为严格，必须在自闭合标签中添加“/”，在 HTML4.01 中，不推荐在自闭合标签中添加“/”。而 HTML5 则较为宽松，添不添加“/”都符合规范。

```
<!-- 不推荐 -->
<link rel="stylesheet" href="./base.css"></link>
<p>ouvenzhang...
```

```
<!-- 推荐 -->
<link rel="stylesheet" href="./base.css">
<p>ouvenzhang...</p>
```

### 👉 使用img的alt属性

为<img>元素加上 alt 属性，alt 属性的内容可以简要描述图片的内容，有利于页面搜索引擎优化，而且对于盲人用户和图像加载失败时的提示也很实用，即支持无障碍阅读和提示，所以要尽量避免 alt 的属性值为空。

```
<!-- 不推荐 -->


<!-- 推荐 -->

```

### 👉 使用label的for属性

为表单内部元素<label>加上 for 属性或者将对应控件放在<label>标签内部，这样在点击<label>标签的时候，同时会关联到对应的 input 或 textarea 上选中，可以增加输入的响应区域。

<!-- 不推荐 -->

```
<label>蓝色</label><input type="radio" name="color" value="#00f">
<label>红色</label><input type="radio" name="color" value="#f00">
```

<!-- 推荐 -->

```
<label for="blue">蓝色</label><input type="radio" id="blue" name="color" value="#00f">
<label for="red">红色</label><input type="radio" id="red" name="color" value="#f00">
```

<!-- 或推荐 -->

```
<label><input type="radio" name="color" value="#00f">蓝色</label>
<label><input type="radio" name="color" value="#f00">红色</label>
```

### 👉 按模块添加注释

在每个大的模块的开始和结束的地方添加起始注释标记，便于开发者识别、维护。

<!-- 新闻列表模块 -->

```
<div id="news" class="g-news"></div>
```

<!-- 新闻列表模块结束 -->

<!-- 排行榜模块 -->

```
<div id="topic" class="g-rank"></div>
```

<!-- 排行榜模块结束 -->

### 👉 标签元素格式

块级元素一般另起一行写。行内元素可以根据情况换行，尽量保证行内元素代码长度不超过一行，否则要考虑另起一行写。HTML 的子元素要尽量相对其父级进行缩进，这样更有层次。

<!-- 不推荐 -->

```
<div><h1>name</h1><p>AAA<em>BBB</em>CCC<span>DDD</span>EEE</p></div>
```

<!-- 推荐 -->

```
<div>
  <h1>name</h1>
  <p>AAA<em>BBB</em>CCC<span>DDD</span>EEE</p>
</div>
```

## 👉 语义化标签

在合适的地方选择语义合适的标签。不要使用被 HTML5 废弃用于样式表现的无语义化标签，如<center>、<font>、<strike>等。

```
<!-- 不推荐移动端使用废弃的标签 -->
<section class="m-news g-mod">
  <header class="m-news-hd">
    <center>头部区域</center>
  </header>
  <div class="m-news-bd">
    <font size="3" color="red">字体标签</font>
  </div>
  <footer class="m-news-ft">
    <strike>底部区域</strike>
  </footer>
</section>

<!-- 推荐移动端语义化布局标签 -->
<section class="m-news g-mod">
  <header class="m-news-hd">
    头部区域
  </header>
  <div class="m-news-bd">
    模块正文
  </div>
  <footer class="m-news-ft">
    底部区域
  </footer>
</section>
```

### 5.1.3 前端CSS规范

#### 👉 CSS引用规范

使用 link 的方式调用外部样式文件，外部样式文件可以复用并能利用浏览器缓存提高加载速度。禁止在标签元素中使用内联样式，否则后期很不容易管理，强烈不建议使用。

```
<!-- 推荐 -->
<link rel="stylesheet" href="main.css">

<!-- 不推荐 -->
<div style="margin: 10px; padding: 10px;"></div>
```

## 👉 样式的命名约定

CSS 类名命名一般由单词、中画线组成，当然也有 BEM（块 block、元素 element、修饰符 modifier，是由 Yandex 团队提出的一种前端命名方法）方案，这里推荐一种规范——所有命名都使用小写，加上 ui-等前缀，表示这个类名只用来控制元素的样式展现。不推荐使用拼音作为样式名，尤其是使用缩写的拼音与英文混合的方式，很让人费解。

```
// 不推荐
.xinwen{}
.XINWEN-list{}
.xinwen-list{}
.ui-xw-ft{}
.news{}
```

尽量不以 info、current、news 等单个单词类名直接作为类名称，单独的一级命名很容易造成冲突覆盖，并且很难理解。

```
// 不推荐
.news .info{}

// 推荐
.ui-news .news-info{}
```

不以模块表现样式来命名，要根据内容来命名。比如 left、right、center、red、black 这类单词命名不允许出现，因为若需求要将某个 left 类名的元素放在右边显示，这就会比较尴尬。推荐使用功能和内容相关联的词汇命名，如<nav>、<news>、<type>、<search>等。

```
// 不推荐
.left{}
.center{}

// 推荐
.ui-search{}
.ui-main{}
```

HTML 标签中的 id 属性一般用于 JavaScript 查询 DOM 使用，书写 CSS 样式时不能用 id 选择器，因为针对 id 的元素样式很难复用。

```
// 不推荐
#news{}

// 推荐
.ui-news{}
```

### 👉 简写方式

单位 0 的缩写。如果属性值为 0，则不需要为 0 加单位。如果是 0 为个数位的小数，前面的 0 可以省略不写。尽量带上分号，否则在后面追加规则时容易因为没写分号而出错。

```
// 不推荐
.ui-news{
  margin: 0px;
  padding: 0px;
  opacity: 0.6
}
```

```
// 推荐
.ui-news{
  margin: 0;
  padding: 0;
  opacity: .6;
}
```

去掉 URL 中引用资源的引号，这是没必要的，影响阅读。

```
// 不推荐
body{
  background-image: url("sprites.png");
}
```

```
// 推荐
body{
  background-image: url(sprites.png);
}
```

颜色值写法，所有的颜色值要使用小写并尽量缩写至 3 位。

```
// 不推荐
body{
  background-color: #FF0000;
}
```

```
// 推荐
body{
  background-color: #f00;
}
```

### 👉 属性书写顺序

CSS 样式书写顺序遵循先布局后内容的规则，即先写元素的布局属性，再写元素的内容属性。

```
// 不推荐
```

```
.ui-news{
    background: #000;
    margin: 10px;
    padding: 10px;
    color: #000;
    width: 500px;
    height:200px;
    float: left;
}
```

// 推荐

```
.ui-news{
    float: left;
    margin: 10px;
    padding: 10px;
    width: 500px;
    height:200px;
    color: #000;
    background: #000;
}
```

这一点比较容易理解。优先布局，常用的布局属性有 position、display、float、overflow 等。内容次之，比如 color、font、text-align。

### 👉 Hack写法

尽可能减少对 CSS Hack 的使用和依赖，可以使用其他的解决方案代替 Hack 的思路。如果必须使用浏览器 Hack，尽量选择稳定、常用并易于理解的书写方式。需要注意的是，目前桌面端浏览器上已经完全舍弃了对 Internet Explorer 7 及以下版本浏览器的兼容性考虑，因此当前 CSS 规则的 Hack 形式写法推荐如下。

```
.ui-news p{
    color:#000;        /* For all */
    color:#111\9;       /* For all IE */
    color:#222\0;       /* For IE8 and later, Opera without Webkit */
    color:#333\9\0;     /* For IE9 and later */
}
```

CSS 规则若要实现在多种浏览器内核上兼容，就要遵循先写私有属性后写标准属性的原则，这样有利于浏览器版本向前兼容。

```
.ui-news{
    -webkit-box-shadow: 1px 1px 5px;
    -moz-box-shadow: 1px 1px 5px;
    -ms-box-shadow: 1px 1px 5px;
    -o-box-shadow: 1px 1px 5px;
    box-shadow: 1px 1px 5px;
```

```
}
```

针对 Internet Explorer, 可以使用条件注释作为预留 Hack 来使用, Internet Explorer 条件注释语法可以如下书写。

```
<!--[if <keywords>? IE <version>?]>
<link rel="stylesheet" href="./hack.css" />
<![endif]-->
```

## 👉 CSS高效实现规范

标签名与 id 或 class 组合的选择器会造成冗余, 而且降低 CSS 的解析速度, 应避免。

```
// 不推荐
div#ui-doc-active.ui-doc{}
```

```
// 推荐:
.ui-doc{}
```

尽量使用简短的 CSS 实现方式, 对于无继承关系的元素使用合并的写法更简洁。

```
// 不推荐:
body{
    margin-top: 10px;
    margin-right: 10px;
    margin-bottom: 10px;
    margin-left: 10px;
}
```

```
// 推荐:
body{
    margin: 10px;
}
```

不同元素之间属性存在继承关系时, 使用分拆方式, 避免继承属性的重复定义。

```
// 不推荐:
.ui-news{
    font: bold 12px arial, sans-serif;
}
.ui-news .new-info{
    font: normal 12px arial, sans-serif;
}
```

```
// 推荐:
.ui-news{
    font: bold 12px arial,sans-serif;
}
.ui-news .new-info{
    font-weight:normal;
```



```
}
```

### 👉 使用预处理脚本编码开发

使用预处理嵌套的方式描述元素之间的层次关系。

```
// 不推荐
.g-box{}
.g-box-hd .xx{}
.g-box-hd .xx .aa{}

// 推荐
.g-box{
  .xx{
    .aa{

    }
  }
}
```

尽可能使用预处理器的高效语法来提高开发效率，如嵌套、变量、嵌套属性、注释、继承等，避免直接使用 CSS 开发。使用 SASS 来编写 CSS 就高效很多。

```
@import "reset.scss";

// 变量赋值与计算能力
$width: 5px + 10px;
$height: 5rem;
$name: box;
$attr: margin;
$content: 'empty text' !default;
$maxWidth: 375px;

// 处理函数
@function getWidth($n){
  @return $n*3rem - 1rem;
}

@mixin mod{
  width: $width;
  height: ($height + 3rem)/2;
  $font-size: 12px;
  $line-height: 20px;
  font: #{$font-size}/#{$line-height};
}

.ui-mod-a{
```

```
// 引用的 mixin 内容将被填充进来
@include mod;
&:hover{
    cursor: pointer;
}
&:after{
    content: $content;
}
.p.#{&name}{
    #{attr}-left: 4px;
}

@media screen and(max-width: $maxWidth){
    #{attr}-left: 8px;
}
}

// 继承的使用, 这里.ui-mod-b 继承了.ui-mod-a 的所有属性, 并且覆写了 width 属性
.ui-mod-b{
    @extend .ui-mod-a;
    @if null {width: $maxWidth;}
    width: getWidth(3);
}
```

### 5.1.4 ECMAScript 5 常用规范

#### 👉 分号

JavaScript 语句后面统一加上分号。以前也有人推荐统一不加分号, 但是加上更容易阅读, 而且更容易和换行语句区分。

```
// 语句之间的关系不容易直接看出
let operate = 1
switch(operate){
    case 1:
        console.log(1)
        break
    case 2:
        console.log(2)
        break
    case 3:
        console.log(3)
        break
    default:
        break
}
```

```
// 语句之间的关系相对清晰一点
let operate = 1;
switch(){
  case 1:
    console.log(1);
    break;
  case 2:
    console.log(2);
    break;
  case 3:
    console.log(3);
    break;
  default:
    break;
}
```

### ✎ 空格

在所有运算符、符号与英文单词之间添加必要的空格，利于开发者阅读。

```
// 不推荐
let a = {
  b :1
};
++ x;
z = x?1:2;
for(let i=0;i<6;i++){
  x++;
}

// 推荐
let a = {
  b: 1
};
++x;
z = x ? 1 : 2;

for (let i = 0; i < 6; i++) {
  x++;
}
```

### ✎ 空行

一般推荐在代码块后保留一行空行，显得块内容层次更加分明，下面的写法是比较推荐的形式。

```
let x = 1;
for (let i = 0; i < 2; i++) {
```

```
    if (true) {
        return false;
    }

    // if 语句后保留空行
    continue;
}

// for 循环语句后保留空行
let obj = {
    foo: function() {
        return 1;
    },

    // 对象方法属性定义后面保留一个空行
    bar: function() {
        return 2;
    }
};
```

## 👉 引号

推荐 JavaScript 字符串最外层统一使用单引号。

```
// 不推荐
let x = "test";

// 推荐
let y = 'foo',
    z = '<div id="z"></div>';
```

## 👉 变量命名

标准变量采用驼峰式命名。常量使用全大写形式命名，并用下画线连接。构造函数首字母大写，jQuery 对象推荐以 “\$” 为开头命名，便于分辨 jQuery 对象和普通对象。

```
// 不推荐
let max_number = 99;
let body = $('body');
let obj_name;

function person(name) {
    this.name = name;
}

// 推荐
const MAX_NUMBER = 99;
const $body = $('body');
```

```
let objName;

function Person(name) {
  this.name = name;
}
```

## 👉 对象

对象属性名不需要加引号。对象属性键值以缩进的形式书写，不要写在同一行。数组、对象属性后不能有逗号，否则部分浏览器可能会解析出错。

```
// 不推荐
let a = { 'b': 1, 'c': 2, };
let b = [1, 2, 3,]

// 推荐
let a = {
  b: 1,
  c: 2
};

let b = [1, 2, 3];
```

## 👉 大括号

程序中的块代码推荐使用大括号包裹，要注意换行，这样更加清晰，而且方便后面扩展增加内容。

```
// 不推荐
if (condition)
  doSomething();

// 推荐
if (condition) {
  doSomething();
  // 添加其他内容
}
```

## 👉 条件判断

尽量不要直接使用 `undefined` 进行变量判断，使用 `typeof` 和字符串 `'undefined'` 对变量类型进行判断。分别用 `===`、`!==` 代替 `==`、`!=` 更加严谨。

```
// 不推荐
if (name == undefined) {
  return false;
}
```

```
// 推荐
if (typeof person === 'undefined') {
  return false;
}
```

### 👉 不要在条件语句或循环语句中声明函数

```
// 不推荐
let name = 'ouven';

if (name) {
  sayHi(name);

  function sayHi (name){
    console.log(`Hi ${name}`);
  }
}
```

```
// 推荐
let name = 'ouven';

if (name) {
  sayHi(name);
}

function sayHi (name){
  console.log(`Hi ${name}`);
}
```

### 👉 一些其他的可选规范参考

for-in 循环里面要尽量含有 hasOwnProperty 的判断，防止访问不存在的对象属性时出错。不要在内置对象的原型上添加方法，如 Array、Date，否则会污染 JavaScript 内置对象。不要在同一个作用域下声明同名变量，这是不安全的 JavaScript 书写方法，严格模式下是禁止使用的。移除声明但未使用过的变量。不要在应该比较的地方赋值。不要像 new function () { ... }、new Object () 等这样使用构造函数。

## 5.1.5 ECMAScript 6+ 参考规范

ECMAScript 5 的通用编码规范在 ECMAScript 6+ 中同样适用，但还是推荐使用 EcmaScript 6+ 更高效的语法来实现与 ECMAScript 5 相同的功能。

### 👉 正确使用ECMAScript 6的变量声明关键字

```
let a = 1;
let A = 2;
const b = 'hello';

{
  let c = 'c';
  const d;    // Uncaught SyntaxError: Missing initializer in const declaration
}

console.log(c); // Uncaught ReferenceError: c is not defined
b = 'world';    // Uncaught TypeError: Assignment to constant letiable.
```

### 👉 字符串拼接使用字符串模板完成

```
// 不推荐
let name = 'ouven';
let str = '<h2>hi, ' + name + '</h2>' +
  '<p>hello, world!</p>' +
  '<p>2016-12-12</p>';

// 推荐
let name = 'ouven';
let str = `<h2>hi, ${name}</h2>
  <p>hello, world!</p>
  <p>2016-12-12</p>
`;
```

### 👉 解构赋值尽量使用一层解构，否则声明变量嵌套太深难以理解

```
// 不推荐
let [[a,b], c] = [[11,22], 33];
let {d, e} = {
  d: {
    key:{
      name: 'ouven'
    }
  },
  e: {
    key:{
      name: 'zhang'
    }
  }
};

// 推荐
let [a, b, c] = [11, 22, 33];
let {d, e} = {
```

```
d: 'ouven',
e: 'zhang',
};
```

### 👉 数组拷贝推荐使用...实现，更加简洁高效

```
const items = [1, 2, 3];
let itemsCopy = [];

// 不推荐
for (let i = 0, len = items.length; i < len; i++) {
  itemsCopy[i] = items[i];
}

// 推荐
itemsCopy = [...items];
```

### 👉 数组循环遍历使用for...of，非必须情况下不推荐使用forEach、map、简单循环

```
const items = [1, 2, 3];

// 不推荐
for (let i = 0, len = items.length; i < len; i++) {
  console.log(items[i]);
}

// 推荐
for (let item of items) {
  console.log(item);
}
```

### 👉 使用ECMAScript 6 的类来代替之前的类实现方式，尽量使用constructor进行属性成员变量赋值

```
// 不推荐
function Foo(name) {
  this.name = name;
  this.sayHi = function(){
    console.log('Hi, ' + this.name);
  }
}

// 推荐
class Foo {
  constructor(name = 'ouven') {
    // constructor
    this.name = name;
  }
}
```



```
sayHi() {
  console.log(`Hi, ${this.name}`);
}
}
```

👉 模块化多变量导出时尽量使用对象解构，不使用全局导出。尽量不要把import和export写在一行

```
// 不推荐
import * as util from './lib/util';

// 推荐
import { time } from './lib/util';

// 不推荐
export default { time } from './lib/util';

// 推荐
import { time } from './lib/util';
export default time;
```

👉 导出类名时，保持模块名称和文件名相同，类名首字符需要大写。

```
// 推荐文件命名为Base.js
class Base {

}

export default Base;
```

👉 生成器中yield进行异步操作时需要使用try...catch包裹，方便对异常进行处理

```
//不推荐
const getNews = function* (){
  this.body= yield render('path/template')
}

// 推荐
const getNews = function* (){
  try{
    this.body= yield render('path/template')
  }catch(e){
    log.error(e);
  }
}
```

### 👉 推荐使用Promise，避免使用第三方库或直接回调，原生的异步处理性能更好而且符合语言规范

```
// 不推荐使用回调方式来处理异步
process('filename', function(data){
  let result = data;
});

// 推荐使用 Promise 来处理
let promise = new Promise(function(resolve, reject){});

promise.then(function(){
  // 成功处理
}, function(){
  // 失败处理
});
```

### 👉 如果不是必须，避免使用迭代器

迭代器 Iterators 性能比较差，对于数组来说大致与 `Array.prototype.forEach` 相当，比不过原生的 `for` 循环，而且使用起来比较麻烦。目前数组遍历提供了 `for...of` 方法，对象遍历提供了 `for...in` 方法，所以非必须情况下还是不建议使用迭代器。

```
const numbers = [1, 2, 3, 4, 5];

// 不推荐
let iterator = numbers[Symbol.iterator]();
let result = iterator.next();
let sum = 0;
while (!result.done) {
  sum += result.value;
  result = iterator.next();
}

// 推荐
let sum = 0;
for (let num of numbers) {
  sum += num;
}
```

### 👉 不要使用统一码，中文的正则匹配和计算较消耗时间，而且容易出问题

### 👉 合理使用Generator，推荐使用async/await，更加简洁

```
// 不推荐
const generator = function* (){
  const numbers = [1, 2, 3, 4, 5];
```

```

    for(let number of numbers){
      yield setTimeout(function(){
        console.log(number);
      }, 3000);
    }
  }
}

let result = generator();
let done = result.next();
while(!done.done){
  done = result.next();
}
console.log('finish');

// 推荐
const asyncFunction = async function (){

  const numbers = [1, 2, 3, 4, 5];
  for(let number of numbers){
    await sleep(3000);
    console.log(number);
  }
}

let result = asyncFunction();
console.log('finish');

```

具体关于 ECMAScript 6+特性的介绍和使用可以参考本书第3章的内容。

### 5.1.6 前端防御性编程规范

前端防御性编程通常不是代码规范中的内容，但却是前端影响网页功能稳定性的一个很重要的因素。简单理解，防御性编程是指通过检测任何可能存在的逻辑异常问题的代码实现，提高脚本执行过程健壮性的一种编程手段。防御性编程要求我们对程序的实现进行更加全面、严谨的考虑。在项目实践中，防御性编程有一些常见的应用场景。

#### 👉 对外部数据的安全检测判断

外部数据可能是从后端返回的内容或用户调用函数时传入的参数等，我们先来看一个模板数据填充渲染的例子。

```
<div>{{ data.userinfo.name }}</div>
```

如果变量 `data` 是后端请求返回的外部数据，那么当 `data` 没有定义 `userinfo` 字段（如 `data={}`）时，前端的模板渲染就会直接报错，或者显示 `data.userinfo.name` 属性未定义，

页面上内容显示可能为空，所以我们尽量不要让这些情况出现，要以一种更安全的方式来展示，并保证模板的填充过程不会出错，例如在数据填充时进行判断，如果没有内容则使用默认的文字代替。

```
// 不推荐
<div>{{ data.userinfo.name }}</div>
```

```
// 推荐
<div>{{ data.userinfo && data.userinfo.name || '默认命名' }}</div>
```

再如我们在 Web 后端对前端提交的数据进行处理时，通常需要先进行检验再进行处理，以免产生安全性漏洞。

```
let id = req.query['id'];

// 如果 id 包含非数字等不合法内容，则提示出错并返回，如果合法才进行查询数据操作
if(!/^d+$/g.test(id)){
  this.body = errorMsg;
}else{
  let sql = `select * where id=${id}`;
  let data = exec(sql);
  this.body = data;
}
```

### 👉 规范化的错误处理

对于常用的 AJAX 请求或长时间文件读写等可能失败的异步操作，需要进行错误情况的处理或异常捕获处理，而不应该被静默，否则一旦出错，用户将得不到正常的提示，对用户体验影响极大。

```
// 不推荐
$.ajax({
  url: 'path/url',
  type: 'get',
  success(data){
    // 数据请求成功后逻辑
  }
});
```

```
// 推荐
$.ajax({
  url: 'path/url',
  type: 'get',
  success(data){
    // 数据请求成功后逻辑
  },
```

```
    error(exception){
        // 请求失败后的逻辑
    }
});

//不推荐
this.body = yield render('path/template');

// 推荐
try{
    this.body = yield render('path/template');
}catch(e){
    log.error(e);
    this.body = yield render('path/error');
}
```

关于防御性编程需要注意的场景还有很多，也是我们编码过程中必须要注意的细节，这些代码严谨性上的考虑可以大大减少代码执行过程中异常出错的概率，而且有利于分析具体的问题。

这一节主要介绍了前端的通用规范、HTML 规范、CSS 规范、ECMAScript 5 规范及 ECMAScript 6+规范、防御性编程的思想，这些都是实践开发中需要关注的。当然，这里列举的不一定全面，也不是说这样就是最好的实践，只是给大家提供一种可选的方案，也让大家明白规范这样制定的原因和好处。大家可以对比下自己的规范特点，取长补短，整理出适合自己或团队的开发编码规范。

## 5.2 前端组件规范

上一节中我们了解了前端的开发编码规范，这一节我们再来了解一下与前端相关的组件规范。

什么是组件规范？为什么需要组件规范？组件规范和开发规范有什么区别和联系呢？首先我们可以认为所谓的组件通常是指采用代码管理中的分治思想，将复杂的项目代码结构拆分成多个独立、简单、解耦合的结构或文件的形式进行分开管理，达到让项目代码和模块更加清晰的目的，而组件规范则是我们进行拆分、组织、管理项目代码方法的一种约定。所以，和开发规范相似，组件规范也是一种约定。不同的是，开发规范关注文件内部代码级别的一致性，组件规范则更关注项目中业务功能模块内容组织的一致性。一定程度上，组件规范包含了开发规范，因为若开发规范不统一，开发出来的组件风格便不一致，组件规范便也无从说起了。组件规范能够帮助我们对功能模块进行统一的约定管理，通过这一约定，任何一个独立的功能模块之间都应该是无耦合并能和其他模块很好对接和组合的。

下面先来看一下目前前端主流的一些组件相关规范：UI（User Interface，用户界面）组件规范、模块化规范、项目组件化设计规范。注意这三者的区别和联系，UI 规范一般指 UI 层设计和实现的规范及统一性，而模块化主要指的是 JavaScript 模块化开发的文件模块封装方式，项目组件规范则指的是实际开发中整个项目业务代码之间的组织形式。

### 5.2.1 UI组件规范

简单来说，UI 组件规范强调了一个网站中所有网页结构层和表现层实现的一致性。多个地方出现的相同按钮样式可以通过公共定义的样式规范类来描述，而不用每个地方都重复书写样式，避免使用不同的代码实现同一个效果。试想如果没有规范的存在，相同作用的按钮有时是红色，有时是绿色，开发维护时就比较难统一处理了。从 Web 前端的角度来看，UI 层的规范能带来一些明显的好处。

- UI 层风格统一化。UI 层风格统一化避免了不同页面的差异化设计风格，能让用户使用 Web 站点的不同网页外观风格是一致的。
- 增加 UI 层复用性。使用 UI 规范的情况下，UI 层代码复用性增强，可以提高开发效率，相同功能的结构和样式不用重复实现。
- 更符合用户的体验习惯。例如红色按钮统一用来表示警告，绿色按钮统一表示安全或成功操作等。
- 增加了开发规范的统一性。遵循统一的规范，避免重复开发，避免产生多种风格的代码。

假如网站中的多个页面含有很多个按钮，通常的做法是把这些按钮分成几类，为每一个按钮元素增加类名，这样写按钮时就不用重复定义了，代码如下。

```
.ui-btn{
  position: relative;
  text-align: center;
  background-color: $button-bg-color;
  background-image: $button-bg-image;
  vertical-align: top;
  color: $button-text;
}

.ui-btn-primary{
  background-color: $button-primary-bg-color;
  border-color: $button-primary-border-color;
  &:not(.disabled):not(:disabled):active,
```

```
&.active{
  background: $button-primary-active-bg;
  border-color: $button-primary-active-bg;
}
}

.ui-btn-warning{
  background-color: $button-warning-bg-color;
  border-color: $button-warning-border-color;
  &:not(.disabled):not(:disabled):active,
  &.active{
    background: $button-warning-active-bg;
    border-color: $button-warning-active-bg;
  }
}
```

这样一来，我们定义不同的 HTML 按钮就可以直接使用了，而且网站任何其他地方使用按钮引用的方法是一致的，就像使用过的 UI 框架一样。不同的是，我们是根据网站页面的特点开发 UI 组件库的。

```
<button class="ui-btn ui-btn-primary">通用按钮</button>
<button class="ui-btn ui-btn-warning">警告按钮</button>
```

那么自己设计 UI 组件规范具体需要遵循哪些原则呢？在实际的团队开发中，UI 组件规范的完成可能需要以下几个方面的协作。

1. UI 设计一致性。在需求开发初始阶段，一般可以通过会议讨论等沟通方式反复确认保证 UI 层设计都是准确的，但尽管如此，如果没有 UI 规范的约束，仍有可能出现某几个相同功能的按钮在不同的页面中样式不同的情况。也就是说，UI 设计层需要统一，相同功能的模块在相同场景下结构层和表现层应该是一致的，否则 UI 规范就没办法实现了。

2. 开发实现的一致性。这就涉及开始说到的编码开发规范了，尽可能让所有的 UI 层实现使用同样的开发规范和方式。例如样式定义、图片引用、命名规范等，就图片引用来说，图标使用 base64 实现还是使用小图片实现，抑或使用 iconfont 实现，都需要统一，不能多种方式混合，否则就增加了 UI 组件的使用复杂度。

所以从开发实现上，如果想要设计实现一个具有通用组件规范的 UI 库（也可能包含了 JavaScript 逻辑），必须考虑以下几个方面的问题。

- 统一的页面布局方案。页面布局使用网格布局还是 REM 方案，是否需要支持响应式，如果是移动端应该怎样适配，这些是需要优先考虑的。
- 基础 UI 结构和样式实现。样式 reset、按钮、图片、菜单、表单等基础结构与样式的

统一化设计实现，可以极大提高页面内容的复用性和开发效率。

- 组件化 UI 结构和样式实现。例如按钮组、字体图标、下拉菜单、输入框组、导航组、面包屑、分页、标签、轮播、弹出框、列表、多媒体、警告框等常用组件的实现。当然网站可能不会一次性用到这么多，但是如果要考虑设计一个通用的 UI 组件库，这些仍然是要考虑的。
- 响应式布局。如果需要支持页面响应式，布局、结构、样式、媒体、javascript 响应式等这些就都要考虑了。具体的实现技术可以参考第 3 章第 7 节介绍的内容。
- 扩展性。如果某个地方用到了比较特殊的样式或逻辑，其他开发者也是可以方便地在原来的代码上进行扩展的。

以 SASS 为例，图 5-1 是目前一个典型的网站基础 UI 组件库的样式设计结构，入口文件 `base.scss` 通过引用不同目录下不同内容的模块编译生成浏览器运行的最终 CSS 样式文件，其中每个 SCSS 文件模块单独管理一个功能的实现：`_variable.scss` 可以专门管理后面可能使用到的变量值、`_mixin.scss` 用于统一管理需要复用的样式规则、`_btn.scss` 用来实现具体多个按钮组件的样式、`_animation.scss` 则保存所有的动画实现。

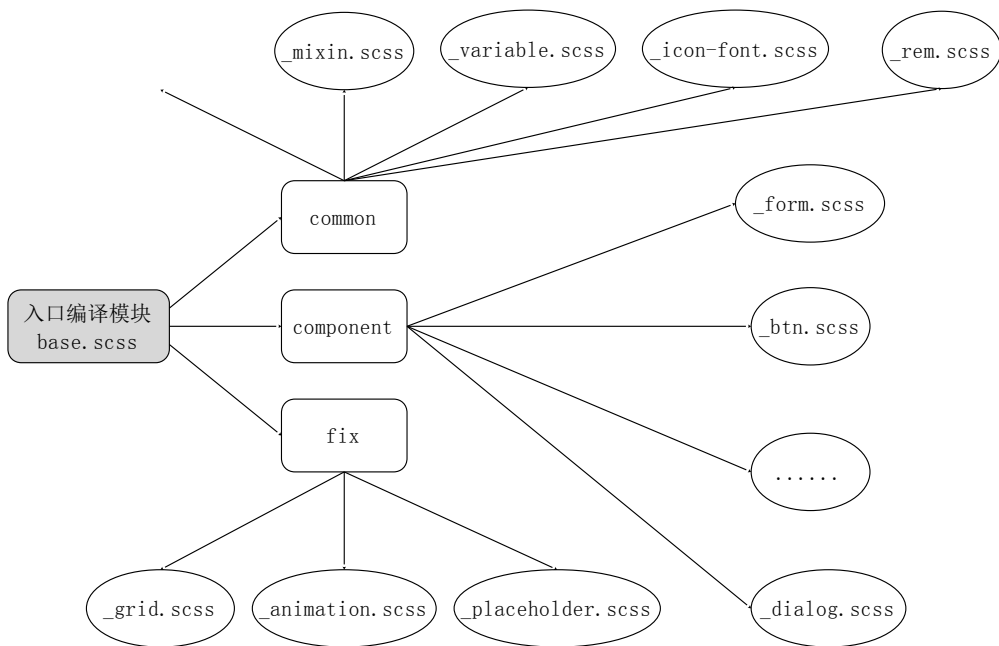


图 5-1 UI 组件表现层管理



通过这样的设计实现 UI 层组件规范就会显得结构很清晰，便于团队并行协作开发，有利于调试时快速定位问题。base.scss 中统一引入其他样式模块的代码如下。

```
@import

"common/reset",
"common/mixin",
"common/variable",
"common/icon-font",
"common/rem",

"fix/grid",
"fix/animation",
"fix/placeholder",

"component/btn",
"component/btn-group",
"component/form",
"component/slider",
"component/tab",
"component/loading",
"component/notice",
"component/dialog",
"component/searchbar";
...
```

### 5.2.2 模块化规范

模块化规范平时讨论比较多，它可以认为是 JavaScript 文件之间相互依赖引用的一种通用语约定，就是按照一定的规范来写 JavaScript 文件，让它可以方便地被其他 JavaScript 文件引用。就规范种类来说，主要包括 AMD（Asynchronous Module Definition，异步模块定义）、CMD（Common Module Definition，通用模块定义）、CommonJS、import/export 等，未来也可能出现其他的规范，下面逐个介绍。

#### AMD

AMD 是运行在浏览器端的模块化异步加载规范，主要以 requireJS 为代表，基本原理是定义 define 和 require 方法异步请求对应的 javascript 模块文件到浏览器端运行。模块执行导出时可以使用函数中的 return 返回结果。

```
// id: 模块的命名，可选参数
// dependencies: 加载的模块依赖列表
// factory: 处理函数，即对 dependencies 加载的模块进行的处理
define(id, dependencies, factory);
```

例如某个主模块 `main` 中引用了两个 JavaScript 文件模块 `mod-A` 和 `mod-B`, 那么主模块调用这两个文件模块并进行初始化的代码如下。

```
// 页面中引用依赖的方式
require('main', ['./mod-A.js', './mod-B.js'], function(A, B){
  A.init();
  B.init();
});

// mod-A.js
define('A', ['zepto'], function($){
  return {
    init(){
      console.log('A')
    }
  }
});

// mod-B.js 的写法
define('B', ['zepto'], function($){
  return {
    init(){
      console.log('B')
    }
  }
});
```

需要注意的是, 主模块处理函数是在 `./mod-A.js` 和 `./mod-B.js` 加载完成并执行结束后才执行的, 即使处理函数中没有用到这两个模块, `./mod-A.js` 和 `./mod-B.js` 一旦被依赖引用就会被加载执行。

## 👉 CMD

CMD 是 `Seajs` 提出的一种模块化规范, 在浏览器端调用类似 `CommonJS` 的书写方式来进行模块引用, 但却不是完全的 `CommonJS` 规范。CMD 遵循按需执行依赖的原则, 只有在用到某个模块的时候才会执行模块内部的 `require` 语句, 同时加载完某个依赖模块文件后并不立即执行, 在所有依赖模块加载完成后进入主模块逻辑, 遇到模块运行语句的时候才执行对应的模块, 这和 `AMD` 是有区别的。

```
define(function(require, exports, module) {});
```

以 `seajs` 为例, 主模块 `main` 中引用了两个 JavaScript 模块 `mod-A` 和 `mod-B`, 调用这两个文件模块并进行初始化的代码如下。

```
seajs.use(['./mod-A.js', './mod-B.js'], function(A, B){
```

```

    A.init();
    B.init();
  });

// mod-A.js 的写法
define(function(require, exports, module) {
  let $ = require('zepto');
  module.exports = {
    init(){
      console.log('A');
    }
  }
});

// mod-B.js 的写法
define(function(require, exports, module) {
  let $ = require('zepto');
  module.exports = {
    init(){
      console.log('B');
    }
  }
});

```

与 AMD 规范不同的是，在引用 `./mod-A.js` 和 `./mod-B.js` 两个文件后，`seajs` 会下载两个模块文件但不会立刻执行，在运行 `init` 方法时才会分别执行两个模块以及处理函数中的动作。

## 👉 CommonJS

CommonJS 是 Node 端使用的 JavaScript 模块化规范，使用 `require` 进行模块引入，并使用 `module.exports` 来定义模块导出。与前面两种方式相比，CommonJS 的写法更加清晰简洁。

```

// main.js
let A = require('./mod-A.js'),
    B = require('./mod-B.js');

A.init();
B.init();

// mod-A.js 的写法
const $ = require('zepto');
module.exports = {
  init(){
    console.log('A');
  }
}

```

```
// mod-B.js 的写法
const $ = require('zepto');
module.exports = {
  init(){
    console.log('B');
  }
}
```

我们一般理解的 `module.exports` 与 `exports` 是同一个对象的不同引用，即 `exports` 可以通俗理解为 `module.exports` 的别名，但是在模块导出时必须使用 `module.exports`。

### 👉 import/export

`import/export` 是 ECMAScript 6 定义的 JavaScript 模块引用方式，是唯一一个遵循 JavaScript 语言标准的模块化规范，在讲解 ECMAScript 6 的时候也重点进行了分析。`import/export` 使用 `import` 引入其他模块，使用 `export` 来进行模块导出。

```
import { initA } from './mod-A.js';
import { initB } from './mod-B.js';
```

```
initA();
initB();
```

```
// mod-A.js 的写法
import Zepto as $ from 'zepto';
```

```
export default {
  initA(){
    console.log('A');
  }
};
```

```
// mod-B.js 的写法
import Zepto as $ from 'zepto';
```

```
export default {
  initB(){
    console.log('B');
  }
};
```

一般不建议直接使用 `import Zepto as $ from 'base'` 进行模块引用，但如果需要导出整个对象时，则必须这么做。

除了了解这些常用的模块化规范外，我们也应该尽量理解模块依赖加载的过程。例如使用 `define` 或 `require` 去读取依赖模块的依赖列表进行引用时，一般模块化支持库会有一个 `baseUrl` 配置或全局的 `id` 配置，这时引用的 `id` 或路径会与 `baseUrl` 进行拼接，计算生成一个绝对或相对路径，例如 `../path/mod-A.js`，然后浏览器创建一个 `<script>` 来加载这个相对路径的文件（Node 端则是通过 `dlopen` 方法进行模块文件内容读取）执行，同时为了避免循环依赖的问题，我们通常会将已经加载的文件标识存入一个缓存数组中，例如 `['../path/a.js']`，下次如果重复引用到同样的文件模块中则无须重复加载，而是直接引用这个模块的返回，然后依次运行加载的模块即可，这样就完成了 JavaScript 模块文件的依赖引用与执行。

关于模块化规范有一个容易误解的地方：很多人认为 AMD 规范只能在浏览器端使用，CommonJS 只能在 Node 端使用。这里要理解的是，模块化规范只是规范，AMD 最早被使用在浏览器端不代表其只能在浏览器端运行，主要还是取决于模块化规范的支持库运行在哪里。例如 `requireJS` 能在浏览器端运行 AMD 规范，在 Node 端也可以实现，不过 Node 端已经有了自己的规范，就不需要去尝试其他方式了。

### 5.2.3 项目组件化设计规范

前端技术发展到现在，为了实现对其复杂的项目进行管理，我们通常使用组件，而且目前实现组件化的方案也已经越来越多：Web Component 组件化、MVVM 框架组件化、基于 Virtual DOM 框架的组件化、直接基于目录管理的组件化等，其中基于每一类方法的实践方案也比较多，下面我们选择一些比较有代表性的实践方案来看看每种组件化的具体设计思路。

#### 📁 Web Component 组件化

前面讲到了 HTML 已经发展到 Web Component 规范阶段，同时也介绍了什么是 Web Component，我们来看一个简单的基于 Web Component 的典型组件化实现方案 Polymer。Polymer 是 Google 在 2013 年的 Google I/O 大会上提出的一个新的 UI 框架，使用了 Web Component 标准，并且针对各种平台的浏览器，Polymer UI 库和组件都具备较好的兼容性。

Polymer 框架的设计主要分成三个层次。

1. 基础层（`platform.js`）：基本实现库。基础层一般都是本地浏览器的 API，例如 `Object.observe()`、事件处理、shadow DOM、自定义元素、HTML 导入、模型驱动视图等。
2. 核心层（`polymer.js`）：可以理解为实现基础层的封装库。

### 3. 元素层：建立在核心层之上的 UI 组件或非 UI 组件。

我们要明白，Web Component 是 Polymer 框架实现的最重要基础，Polymer 的设计是面向组件的，使用 HTML imports 技术来加载组件，定义的元素也可以没有用户界面。我们以自定义的某个组件为例，来看看 Polymer 的具体实现思路。

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>图文组合插件</title>
  <script src="../../components/platform/platform.js"></script>
  <!-- import 引入组件内容 -->
  <link href="./image.html" rel="import" />
</head>
<body>
  <!-- 这里注册生成了一个 shadow host 为<x-image>的 DOM 元素 -->
  <x-image src="./image.jpg" width="290" height="160"></x-image>
</body>
</html>
```

图 5-2 为 Polymer 组件化规范样例，显示了一个带浮层文字的图片显示组件。Polymer 支持双向的数据绑定，更新数据模型会反映在 DOM 上，而 DOM 上的用户输入也会立即修改数据模型上的数据。对于 Polymer 元素来说，对应数据模型始终是元素本身。



图 5-2 Polymer 组件化规范样例

```
<x-image name="x-image">
  <style>
    div{
      color: red;
    }
  </style>
  <template>
    <div class="x-image-section">
      <span class="x-image-image">
        <img class='image' src="" alt="image" height="200">
      </span>
    </div>
  </template>
</x-image>
```

```

    </span>
    <span class="x-image-text">{{text}}</span>
</div>
</template>
<script>
  Polymer({
    is: 'x-image',
    properties: {
      text: '带文字描述的图片'
    }
  });
</script>
</x-image>

```

元素数据模型可以用如下方式直接修改。

```
document.querySelector('x-image').text = '修改的文字描述图片';
```

关于 Polymer 应用的内容还有很多，这里介绍的是 Web Component 这种组件化实现的技术原理和使用。实际上，Polymer 自提出后到现在并没有得到很广泛的实践，但是其遵循 Web Component 规范的这一实践思路被越来越多的组件化框架借鉴。

## 📌 MVVM框架组件化

MVVM 组件化在实现上比较有吸引力，但其实非常简单，其基本思路是将页面中的模块按照元素来划分，并将与这个模块相关的 MVVM 描述语法、CSS 样式、执行脚本放在同一个文件里进行引用。

```

<style>
  body{
    background-color: #ccc;
  }
</style>
<template>
  <h2>{{ text }}</h2>
  <button q-on="clickEvent(this)"></button>
</template>
<script>
  let $ = require('jQuery');
  module.exports = {
    data: {
      text: '这是一段描述'
    },
    events: {
      clickEvent() {
        console.log(this.model.data.text);
      }
    }
  }

```

```
};  
</script>
```

一般推荐每个组件以单个文件的形式来引入模块相关内容，然后通过构建或动态解析的方式动态获取该组件包含的 CSS、HTML、JavaScript 脚本到页面中使用，而且组件内容书写的方式也往往和 Polymer 类似，前端的三层结构要绑在一起管理。相比之下，和 Polymer 主要的区别在于其使用的不是自定义元素组件，而是带有 MVVM 语法的 HTML 标签。所以目前主流 MVVM 组件化的设计思路和 Polymer 的实现思路基本是类似的。

### 📌 Virtual DOM的组件化方案

之前讲到，Virtual DOM 的出现更多情况下是为了改善 MVVM 的 DOM 性能。以 reactjs 为例，虽然 reactjs 改变了我们对 DOM 操作的原有理解，但是在组件化设计实现方面，它使用的仍是和 Polymer 类似的组织管理方式，所不同的是将 HTML 的结构描述换了另一种语法形式且 JavaScript 的调用 API 不同。

```
const React = require('react');  
const ReactDOM = require('react-dom');  
const styles = require('./mod.css');  
  
let TextImage= React.createClass({  
  // 组件Virtual DOM 描述语法  
  clickEvent(){  
    console.log(this.props.text);  
  },  
  
  render() {  
    return (  
      <div>  
        <h2>{ this.props.text }</h2>  
        <button onClick="{ this.clickEvent }"></button>  
      </div>  
    );  
  }  
});  
  
export default TextImage;
```

这是个简单的例子，和 MVVM 的方式相比，其组件内容结构和使用方式仍是类似的，不同点只在于页面逻辑的执行过程和框架的使用，但这并不是和组件化规范设计相关的内容。

### 📌 基于目录管理的通用组件化实践

目前主流框架的组件化实践方案虽然很多且表现形式不相同，但是实现设计的基本思路还



是一样的，即和 Polymer 保持一致：将页面的三层结构内容绑定到一起作为一个独立的组件存在，然后通过解析应用到页面中执行。当然，如果将三层结构直接绑到一起可能会有些问题，例如 CSS 解析失败会导致整个模块加载失败，结构比较混淆时三层结构不分离，不容易进行团队协作。

既然如此，我们使用一个目录的形式来分开管理前端页面的三层结构岂不更加灵活方便吗？对组件的三层结构进行文件划分，各个文件负责自己的功能部分，然后在构建生成的时候进行组件中不同类文件内容的打包处理。

```
/component
index.es      // 组件逻辑处理
index.scss    // 组件预处理脚本
index.html    // 组件 HTML 结构
/img         // 组件可能用到的背景图片
```

按照这个思路，组件三层结构的每一部分都解耦并且与使用的框架无关。例如 JavaScript 部分可以任意选择 ECMAScript 6+或 typescript 进行开发，或者是使用不同的框架编写。CSS 预处理也能任意选择团队熟知的预处理语法。HTML 结构层可以自由选择普通前端模板、MVVM 的语法结构或者 Virtual DOM 的描述语法实现，只要脚本支持即可。

总结来看，尽管这里讲了四种不同形式的组件化实践方案，但它们的设计思路是一致的，都是前端三层结构的组织规范设计。尽管目前不同主流框架实现组件化的思路都有各自的亮点，但并没有太多差异的地方。因此更推荐使用组件目录式的组件化设计规范，通过目录来管理组件，而不是通过文件或依赖具体的某个框架。前面的三种方式也可以很容易地改成基于目录规范的形式，这样就可以做到三层结构开发分离并且更加灵活地控制，更有利于前端项目的开发和维护。

我们再来看一下设计一个高效的组件化规范应该解决哪些问题。

- 组件之间独立、松耦合。组件之间的 HTML、JavaScript、CSS 之间相互独立，尽量不重复，相同部分通过父级或基础组件来实现，最大限度减少重复代码。
- 组件间嵌套使用。组件可以嵌套使用，但嵌套后仍然是独立、松耦合的。
- 组件间通信。主要指组件之间的函数调用或通信，例如 A 组件完成某个操作后希望 B 组件执行某个行为，这种情况就可以使用监听或观察者模式在 B 组件中注册该行为的事件监听或加入观察者，然后选择合适的时机在 A 组件中触发这个事件监听或通知观察者来触发 B 组件中的行为操作，而不是在 A 组件中直接拿到 B 组件的引用并直接进

行操作，因为这样组件之间的行为就会产生耦合。

- 组件公用部分设计。组件的公用部分应该被抽离出来形成基础库，用来增加代码的复用性。
- 组件的构建打包。构建工具能够自动解析和打包组件内容。
- 异步组件的加载模式。在移动端，通常考虑到页面首屏，异步的场景应用非常广泛，所有异步组件不能和同步组件一起处理。这时可以将异步组件区别于普通组件的目录存放，并在打包构建时进行异步打包处理。
- 组件继承与复用性。对于类似的组件要做到基础组件复用来减少重复编码。
- 私有组件的统一管理。为了提高协作效率，可以通过搭建私有源的方式来统一管理组件库，例如使用包管理工具等。但这点即使在大的团队里面也很难实施，因为业务组件的实现常常需要定制化而且经常变更，这样维护组件库成本反而更大，目前可以做的是将公用的组件模块使用私有源管理起来。
- 根据特定场景进行扩展或自定义。如果当前的组件框架不能满足需求，我们应该能够很便捷地拓展新的框架和样式，这样就能适应更多的场景需求。比如在通过目录管理组件的方案下，既可以使用 MVVM 框架进行开发，也可以使用 Virtual DOM 框架进行开发，但要保持基本的规范结构不变。

关于组件规范，我们主要了解了 UI 组件规范、模块化规范和组件设计规范。对于组件设计规范，我们要认识其本质——前端三层结构的设计和组织形式。个人推荐使用目录的组件规范设计形式来更加灵活地管理和组织代码。

## 5.3 自动化构建

在现代前端工程开发中，自动化构建已经成为一个必不可少的部分，本节我们就来讨论一下前端自动化构建方面的知识。

目前的构建工具多种多样，设计的思路也略有不同，但是整体的实现原理却是基本一致的。这里我们仍以讲述构建工具和流程原理性的内容为主，目的是希望你不仅掌握如何使用现在的构建工具，同时也能了解构建工具自动完成构建的过程。

提到前端自动化构建工具，我们可以追溯到软件开发时代的 IDE（Integrated Development

Environment，集成开发环境）。IDE 在软件编译运行阶段将软件所需要的代码、资源、图片等打包成一个可以独立运行的软件安装包，然后在不同平台上安装运行。前端开发中是没有这样的 IDE 的，因为前端代码不需要软件编译。举一个简单的例子，我们在一个页面中使用了多个背景图片，但是想把这几个背景图片请求合成一个图片（俗称雪碧图），以前我们可能会手动把这些图片放在一个大背景图片中，然后通过元素的背景图偏移量来实现多个元素对它的引用。后来，页面又添加了一个图片，我们就在这个原有合成的大图片中新添加了一个图片。很多人应该有过这样的经历，这样很麻烦，于是我们希望能有一个像软件 IDE 这样的工具，对代码进行分析，把引用的各种资源打包统一处理，自动输出成为理想的结构。合并多个背景图片只是其中一个场景，这一类问题就是前端构建工具需要解决的，某种意义上，前端构建工具很像软件开发 IDE 的编译打包处理模块。

5.3.1 自动化构建的目的

前端构建工具的作用可以认为是对源项目文件或资源进行文件级处理，将文件或资源处理成需要的最佳输出结构和形式。在处理过程中，我们可以对文件进行模块化引入、依赖分析、资源合并、压缩优化、文件嵌入、路径替换、生成资源包等多种操作，这样就能完成很多原本需要手动完成的事情，极大地提高开发效率。

5.3.2 自动化构建原理

主流的构建工具虽然种类较多，但原理相通。下面我们来看看目前主流的构建工具的实现原理和过程。首先要明确的是，自动化构建是基于项目代码文件级的分析处理，下面以一个通用构建工具实现的文件处理流程为例向大家介绍。

如图 5-3 所示，构建的流程主要分成 7 个基本步骤（不同的构建工具各有差异，但基本原理是类似的）：读取入口文件→分析模块引用→按照引用加载模块→模块文件编译处理→模块文件合并→文件优化处理→写入生成目录。

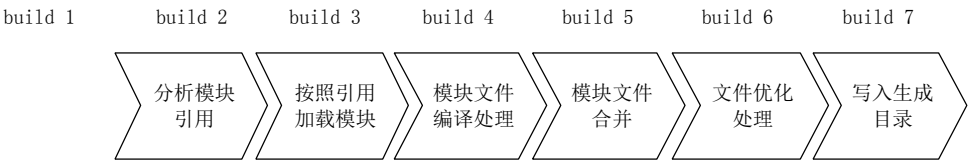


图 5-3 构建原理流程

```
<!-- index.html -->
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <!--style-->
</head>
<body>
  <mod-A></mod-A>
  <mod-B></mod-B>
  <script src="main.js"></script>
  <!--script-->
</body>
</html>
```

其中模块 A 和模块 B 组件对应的目录文件如下。

```
index.es
index.html
index.scss
img/
```

以上面这一个入口文件的 index.html 源文件为例。这个入口页面文件 index.html 中含有 A 和 B 两个模块，模块 A、B 组件遵循统一的模块规范：每个组件都包含 JavaScript、SCSS、HTML 文件和 img 目录。我们希望构建后将模块 A 和模块 B 的内容全部引用到页面上，CSS 和 JavaScript 的脚本资源也经过压缩编译处理，而且最后打包后的资源引用达到最优预上线的状态。根据上面的构建处理流程，我们将构建任务分成 7 个阶段（分别命名为 build1~build7），具体如下。

- build1 读取入口文件阶段。构建工具会读取 index.html 源文件到一个字符串 Buffer（或者文件对象）中。
- build2 分析模块引用阶段。根据特定的标识（例如 mod-开头的自定义标签）分析出页面字符串 Buffer 中含有的两个模块 A 和模块 B 的引用。
- build3 按照引用加载模块文件阶段。进入模块 A、B 目录中读取模块 A 和模块 B 包含的 HTML、SCSS、JavaScript 文件。
- build4 模块文件编译阶段。进行对应的脚本转译（转译的工具都是通过插件完成的）和依赖分析，例如将 ECMAScript 6+脚本转译成 ECMAScript 5 脚本模块引入、将 SCSS 文件预处理为 CSS 等。该阶段完成后，构建工具将生成编译后的代码字符串 Buffer。
- build5 模块文件合并阶段。将所有 JavaScript、CSS 代码字符串 Buffer 写入一个新的字符串 Buffer 中，将模块 A、B 的 HTML 字符串 Buffer 插入 index.html 的字符串 Buffer 中，生成线上域名路径 <http://www.domain.com/dist/css/main.css> 和

`http://www.domain.com/dist/js/main.js`, 将路径名插入到 `index.html` 字符串 `Buffer` 中代替注释 `<!--style-->` 和 `<!--script-->` 的位置, 表示 CSS 和 JavaScript 脚本引用的最终路径。

- **build6 文件优化处理阶段。**将合并后的 JavaScript、CSS 代码字符串 `Buffer` 进行优化, 例如去注释、压缩等。或将 CSS 引用的多个单张背景图合成一张大图, 这个阶段的压缩合并处理也都是可以通过不同插件来优化完成的。
- **build7 阶段。**将优化完成的字符串 `Buffer` 写入到配置好的输出目录中, 将文件命名为 `main.js` 和 `main.css`, 对 HTML 字符串 `Buffer` 进行去除注释等优化处理, 最后写入到输出目录中。至此, 整个构建流程完成。分析处理后入口 HTML 文件可能如下, 而引用的 CSS 和 JavaScript 文件都是转译优化后的代码文件。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <link rel="stylesheet" href="http://www.domain.com/dist/css/main.css">
</head>
<body>
  <div class="mod-A">
    ...
  </div>
  <div class="mod-B">
    ...
  </div>
  <script src="http://www.domain.com/dist/js/main.js"></script>
</body>
</html>
```

处理的文件以字符串 `Buffer` 或文件对象的形式存在于整个过程中, 最终生成文件的规则一般是按照用户自定义的配置来完成的。不同阶段的处理流程一般可以通过第三方插件来实现, 例如上面提到模块化封装、依赖合并、压缩优化、文件嵌入、路径替换、生成资源包等都可以借助插件来做, 必要的时候也会自己编写插件。

从某种意义上来说, 构建流程中代码资源文件就像是工厂生产线的产品一样, 经过多个加工机器进行处理加工, 最后打包封装生成想要的产品。早期也有通过不断读写文件处理方式实现的构建工具, 即每经过一次处理就将文件写到磁盘的某个临时目录下, 处理下一个插件时再从这个目录中读取出来, 但这种方式由于频繁地对磁盘进行读写, 因此构建速度较慢, 目前已经很少使用了。

### 5.3.3 构建工具设计的问题

通过上面的分析，我们对构建工具的流程和原理有了较直接的认识，并可以使用基础的工具搭建自己理想的构建环境。那么在现代前端实际开发中通常希望构建工具帮我们处理哪些问题呢？

#### 📁 模块分析引入

目前前端开发模式基本已经进入组件化开发阶段，组件化实现的方式也比较多，所以首先希望构建工具能自动帮我们完成对组件模块的引入和分析，例如自动分析 `require` 或 `import` 的模块进行合并打包。这样在开发过程中就可以专注于组件本身，而不用关心组件模块引用最终是怎么被构建打包的。

重点了解一下 JavaScript 组件模块文件的依赖分析过程，以 `require` 的引用方式为例。

1. 从入口模块开始分析 `require` 函数调用依赖。
2. 根据依赖生成 JavaScript AST (Abstract Syntax Tree, 抽象语法树, 是将 JavaScript 代码映射成一个树形结构的 JSON 对象树)。
3. 根据 AST 找到每个模块的模块名。
4. 得到每个模块的依赖关系，生成一个依赖字典。
5. 根据模块化引用机制包装每个模块，传入依赖字典以及 `import` 或 `require` 函数，生成执行的 JavaScript 代码。

#### 📁 模块化规范支持

目前，前端开发中使用 ECMAScript 6+ 标准实现的项目居多，但大多数情况都是通过构建工具插件将其转译成 ECMAScript 5 语法代码在浏览器中运行的。这里涉及多种 JavaScript 模块化规范之间处理转换的问题，好的构建工具应该尽可能支持较多种类模块化规范进行打包，这样我们就不用考虑模块化规范不统一的问题了。

#### 📁 CSS编译、自动合并图片

CSS 的预处理、图片引用的分析、内联图片资源、自动合并雪碧图等功能在一个理想的构建工具里都是必不可少的。如果构建工具能自动移除解析后重复冗余的 CSS 规则就更完美了。

#### 📁 HTML、JavaScript、CSS资源压缩优化

为了尽可能让线上资源的体积最小化，发布到线上的文件通常需要经过构建工具的压缩优化处理，例如 HTML 内注释和多余空格的压缩、JavaScript 的 `uglify` 操作、CSS 的压缩等，这

些都是我们希望构建工具自动完成的。

### 🔗 HTML路径分析替换

在开发过程中，为了方便开发调试处理，我们通常使用相对路径来进行文件的引用。但是项目开发完成上线后，静态资源会发布到绝对路径甚至不同的域名 CDN 路径上，这就需要在上线前对文件路径进行替换，所以我们希望构建工具也能自动完成这一工作，即提供将文件相对路径自动替换成绝对路径或线上 CDN 路径的能力。

### 🔗 区分开开发和上线目录环境

为了方便开发和调试，希望浏览器在使用构建工具开发时能加载未压缩处理的代码文件，但是准备上线前需要对文件进行压缩优化处理，这就要求构建工具能区开发 and 线上环境的不同资源。我们常常希望能够在配置构建任务时指明开发和发布资源目录来满足不同场景下的需要。

### 🔗 异步文件打包方案

异步文件加载的场景很常见，尤其是在移动端，我们通常将首屏的内容优先展示，然后滚动异步加载后面的内容。为了保证首屏加载的资源最小，非首屏的内容都希望通过 JavaScript 来异步渲染，这就需要构建工具能将非首屏的组件打包成异步资源，以按需或异步的方式加载。同时我们又不希望在开发过程中太关注异步组件和同步组件的区别，所以通常将异步组件放在异步的目录里进行单独打包或加入特殊的标识，不过也需要构建工具支持才行。

### 🔗 文件目录白名单设置

为了加快项目代码构建的处理速度，建议提供一些配置绕过不需要处理的文件目录，否则文件目录较多的情况下构建处理速度就会比较慢。

除了这些，我们还可以配置构建工具来完成更多的事情，具体可以根据特殊需要来选择使用，所以通常构建工具也应该是可以扩展的，通过配置更多的插件来共同完成项目自动化处理中的任务。

这一节主要讲解了构建工具的工作流程和原理，介绍得相对比较理论化，希望大家结合自己使用的构建工具去理解消化。对于构建工具的选择，读者可以比较当下主流的构建工具并选择一个合适的使用。构建工具仍在不断更新变化，但无论如何，构建的基本原理是确定的，它的最终目的仍是自动化完成一些事情，提高开发的效率。

## 5.4 前端性能优化

前端性能优化是一个很宽泛的概念，本书前面的部分也多多少少提到一些前端优化方法，这也是我们一直在关注的一件重要事情。配合各种方式、手段、辅助系统，前端优化的最终目的都是提升用户体验，改善页面性能，我们常常竭尽全力进行前端页面优化，但却忽略了这样做的效果和意义。先不急于探究前端优化具体可以怎样去做，先看看什么是前端性能，应该怎样去了解和评价前端页面的性能。

通常前端性能可以认为是用户获取所需要页面数据或执行某个页面动作的一个实时性指标，一般以用户希望获取数据的操作到用户实际获得数据的时间间隔来衡量。例如用户希望获取数据的操作是打开某个页面，那么这个操作的前端性能就可以用该用户操作开始到屏幕展示页面内容给用户的这段时间间隔来评判。用户的等待延时可以分成两部分：可控等待延时和不可控等待延时。可控等待延时可以理解为能通过技术手段和优化来改进缩短的部分，例如减小图片大小让请求加载更快、减少 HTTP 请求数等。不可控等待延时则是不能或很难通过前后端技术手段来改进优化的，例如鼠标点击延时、CPU 计算时间延时、ISP (Internet Service Provider, 互联网服务提供商) 网络传输延时等。所以要知道的是，前端中的所有优化都是针对可控等待延时这部分来进行的，下面来了解一下如何获取和评价一个页面的具体性能。

### 5.4.1 前端性能测试

获取和衡量一个页面的性能，主要可以通过以下几个方面：Performance Timing API、Profile 工具、页面埋点计时、资源加载时序图分析。

#### 📌 Performance Timing API

Performance Timing API 是一个支持 Internet Explorer 9 以上版本及 WebKit 内核浏览器中用于记录页面加载和解析过程中关键时间点的机制，它可以详细记录每个页面资源从开始加载到解析完成这一过程中具体操作发生的时间点，这样根据开始和结束时间戳就可以计算出这个过程所花的时间了。

图 5-4 为 W3C 标准中 Performance Timing 资源加载和解析过程记录各个关键点的示意图，浏览器中加载和解析一个 HTML 文件的详细过程先后经历 unload、redirect、App Cache、DNS、TCP、Request、Response、Processing、onload 几个阶段，每个过程开始和结束的关键时间戳浏览器已经使用 performance.timing 来记录了，所以根据这个记录并结合简单的计算，我们就可以得到页面中每个过程所消耗的时间。



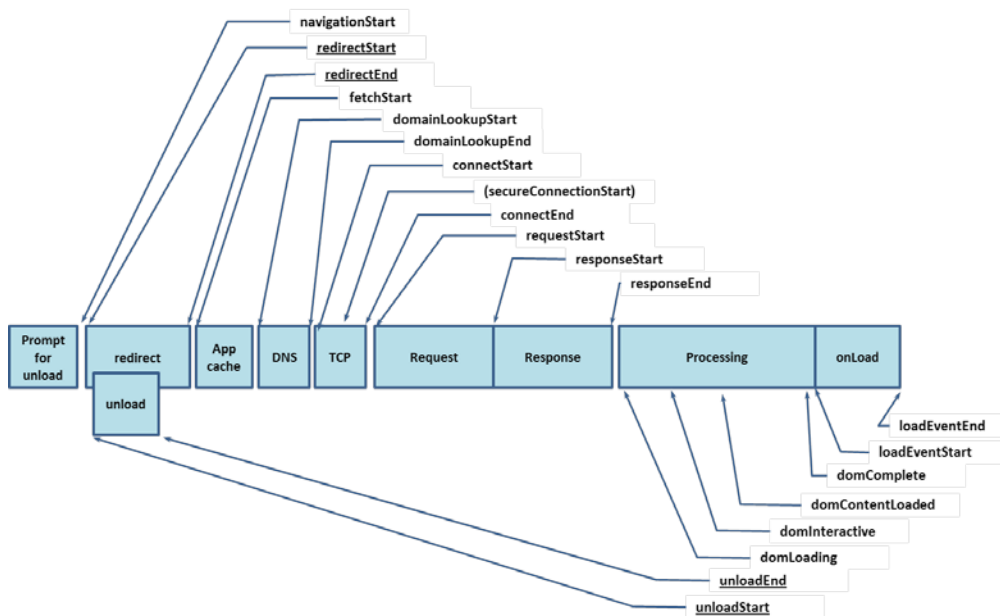


图 5-4 performance API 关键时间点记录

```
function performanceTest(){
    let timing = performance.timing,
        readyStart = timing.fetchStart - timing.navigationStart,
        redirectTime = timing.redirectEnd - timing.redirectStart,
        appcacheTime = timing.domainLookupStart - timing.fetchStart,
        unloadEventTime = timing.unloadEventEnd - timing.unloadEventStart,
        lookupDomainTime = timing.domainLookupEnd - timing.domainLookupStart,
        connectTime = timing.connectEnd - timing.connectStart,
        requestTime = timing.responseEnd - timing.requestStart,
        initDomTreeTime = timing.domInteractive - timing.responseEnd,
        domReadyTime = timing.domComplete - timing.domInteractive,
        loadEventTime = timing.loadEventEnd - timing.loadEventStart,
        loadTime = timing.loadEventEnd - timing.navigationStart;

    console.log('准备新页面时间耗时: ' + readyStart);
    console.log('redirect 重定向耗时: ' + redirectTime);
    console.log('Appcache 耗时: ' + appcacheTime);
    console.log('unload 前文档耗时: ' + unloadEventTime);
    console.log('DNS 查询耗时: ' + lookupDomainTime);
    console.log('TCP 连接耗时: ' + connectTime);
    console.log('request 请求耗时: ' + requestTime);
    console.log('请求完毕至 DOM 加载: ' + initDomTreeTime);
    console.log('解析 DOM 树耗时: ' + domReadyTime);
    console.log('load 事件耗时: ' + loadEventTime);
}
```

```
console.log('加载时间耗时: ' + loadTime);
}
```

通过上面的时间戳计算可以得到几个关键步骤所消耗的时间，对前端有意义的几个过程主要是解析 DOM 树耗时、load 事件耗时和整个加载过程耗时等，不过在页面性能获取时我们可以尽量获取更详细的数据信息，以供后面分析。除了资源加载解析的关键点计时，performance 还提供了一些其他方面的功能，我们可以根据具体需要进行选择使用。

```
performance.memory // 内存占用的具体数据
performance.now() // performance.now()方法返回当前网页自 performance.timing 到现在的时间，可以精确到微秒，用于更加精确的计数。但实际上，目前网页性能通过毫秒来计算就足够了
performance.getEntries() // 获取页面所有加载资源的 performance timing 情况。浏览器获取网页时，会对网页中每一个对象（脚本文件、样式表、图片文件等）发出一个 HTTP 请求。performance.getEntries 方法以数组形式返回所有请求的时间统计信息
performance.navigation // performance 还可以提供用户行为信息，例如网络请求的类型和重定向次数等，一般都存放在 performance.navigation 对象里面
performance.navigation.redirectCount // 记录当前网页重定向跳转的次数
```

参考资料：<https://www.w3.org/TR/resource-timing/>。

## 📁 Profile工具

Performance Timing API 描述了页面资源从加载到解析各个阶段的执行关键点时间记录，但是无法统计 JavaScript 执行过程中系统资源的占用情况。Profile 是 Chrome 和 Firefox 等标准浏览器提供的一种用于测试页面脚本运行时系统内存和 CPU 资源占用情况的 API，以 Chrome 浏览器为例，结合 Profile，可以实现以下几个功能。

1. 分析页面脚本执行过程中最耗资源的操作
2. 记录页面脚本执行过程中 JavaScript 对象消耗的内存与堆栈的使用情况
3. 检测页面脚本执行过程中 CPU 占用情况

使用 console.profile() 和 console.profileEnd() 就可以分析中间一段代码执行时系统的内存或 CPU 资源的消耗情况，然后配合浏览器的 Profile 查看比较消耗系统内存或 CPU 资源的操作，这样就可以有针对性地进行优化了。

```
console.profile();
// TODOS, 需要测试的页面逻辑动作
for(let i = 0; i < 100000; i++){
  console.log(i * i);
}
console.profileEnd();
```

## 👉 页面埋点计时

使用 **Profile** 可以在一定程度上帮助我们分析页面的性能，但缺点是不够灵活。实际项目中，我们不会过多关注页面内存或 CPU 资源的消耗情况，因为 JavaScript 有自动内存回收机制。我们关注更多的是页面脚本逻辑执行的时间。除了 **Performance Timing** 的关键过程耗时计算，我们还希望检测代码的具体解析或执行时间，这就不能写很多的 `console.profile()` 和 `console.profileEnd()` 来逐段实现，为了更加简单地处理这种情况，往往选择通过脚本埋点计时的方式来统计每部分代码的运行时间。

页面 JavaScript 埋点计时比较容易实现，和 **Performance Timing** 记录时间戳有点类似，我们可以记录 JavaScript 代码开始执行的时间戳，后面在需要记录的地方埋点记录结束时的时间戳，最后通过差值来计算一段 HTML 解析或 JavaScript 解析执行的时间。为了方便操作，可以将某个操作开始和结束的时间戳记录到一个数组中，然后分析数组之间的间隔就得到每个步骤的执行时间，下面来看一个时间点记录和分析的例子。

```
let timeList = [];
function addTime(tag){ timeList.push({"tag":tag,"time":+new Date}); }

addTime("loading");

timeList.push({"tag":"load","time": +new Date()});
// TODOS, load 加载时的操作
timeList.push({"tag":"load","time": +new Date()});

timeList.push({"tag":"process","time": +new Date()});
// TODOS, process 处理时的操作
timeList.push({"tag":"process","time": +new Date()});

parseTime(timeList); // 输出{load: 时间毫秒数, process: 时间毫秒数}

function parseTime(time){
  let timeStep = {},
      endTime;
  for(let i = 0,len = time.length; i < len; i ++){
    if(!time[i]) continue;

    endTime = {};
    for(let j = i+1; j < len; j++){
      if(time[j] && time[i].tag == time[j].tag){
        endTime.tag = time[j].tag;
        endTime.time = time[j].time;
        time[j] = null;
      }
    }
  }
}
```

```
    if(endTime.time >= 0 && endTime.tag){
        timeStep[endTime.tag] = endTime.time - time[i].time;
    }
}
return timeStep;
}
```

这种方式常常在移动端页面中使用，因为移动端浏览器 HTML 解析和 JavaScript 执行相对较慢，通常为了进行性能优化，我们需要找到页面中执行 JavaScript 耗时的操作，如果将关键 JavaScript 的执行过程进行埋点计时并上报，就可以轻松找出 JavaScript 执行慢的地方，并有针对性地进行优化。

### 资源加载时序图

我们还可以借助浏览器或其他工具的资源加载时序图来帮助分析页面资源加载过程中的性能问题。这种方法可以粗粒度地宏观分析浏览器的所有资源文件请求耗时和文件加载顺序情况，如保证 CSS 和数据请求等关键性资源优先加载，JavaScript 文件和页面中非关键性图片等内容延后加载。如果因为某个资源的加载十分耗时而阻塞了页面的内容展示，那就要着重考虑。所以，我们需要通过资源加载时序图来辅助分析页面上资源加载顺序的问题。

图 5-5 为使用 Fiddler 获取浏览器访问地址 <http://www.jixianqianduan.com> 时的资源加载时序图。根据此图，我们可以很直观地看到页面上各个资源加载过程所需要的时间和先后顺序，有利于找出加载过程中比较耗时的文件资源，帮助我们有针对性地进行优化。

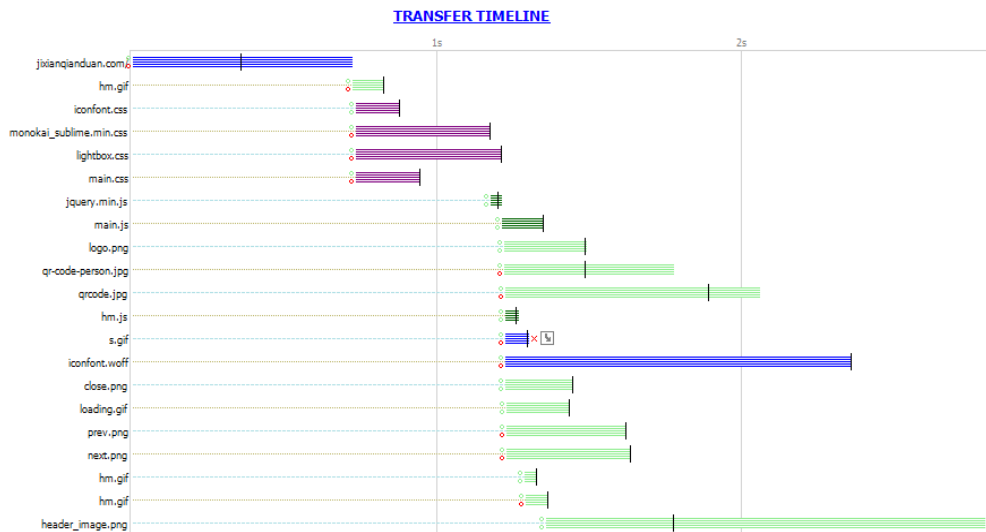


图 5-5 页面加载文件资源时序图

## 5.4.2 桌面浏览器前端优化策略

通过性能测速和分析，我们基本可以获取收集到页面上大部分的具体性能数据，如何根据这些数据采取适当的方法和手段对当前的页面进行优化呢？目前来看，前端优化的策略很多，如 YSlow（YSlow 是 Yahoo 发布的一款 Firefox 插件，可以对网站的页面性能进行分析，提出对该页面性能优化的建议）原则等，总结起来主要包括网络加载类、页面渲染类、CSS 优化类、JavaScript 执行类、缓存类、图片类、架构协议类等几类，下面逐一介绍。

### 👉 网络加载类

#### 1. 减少 HTTP 资源请求次数

在前端页面中，通常建议尽可能合并静态资源图片、JavaScript 或 CSS 代码，减少页面请求数和资源请求消耗，这样可以缩短页面首次访问的用户等待时间。通过构建工具合并雪碧图、CSS、JavaScript 文件等都是为了减少 HTTP 资源请求次数。另外也要尽量避免重复的资源，防止增加多余请求。

#### 2. 减小 HTTP 请求大小

除了减少 HTTP 资源请求次数，也要尽量减小每个 HTTP 请求的大小。如减少没必要的图片、JavaScript、CSS 及 HTML 代码，对文件进行压缩优化，或者使用 gzip 压缩传输内容等都可以用来减小文件大小，缩短网络传输等待时延。前面我们使用构建工具来压缩静态图片资源以及移除代码中的注释并压缩，目的都是为了减小 HTTP 请求的大小。

#### 3. 将 CSS 或 JavaScript 放到外部文件中，避免使用<style>或<script>标签直接引入

在 HTML 文件中引用外部资源可以有效利用浏览器的静态资源缓存，但有时候在移动端页面 CSS 或 JavaScript 比较简单的情况下为了减少请求，也会将 CSS 或 JavaScript 直接写到 HTML 里面，具体要根据 CSS 或 JavaScript 文件的大小和业务的场景来分析。如果 CSS 或 JavaScript 文件内容较多，业务逻辑较复杂，建议放到外部文件引入。

```
<link rel="stylesheet" href="//cdn.domain.com/path/main.css">
...
<script src="//cdn.domain.com/path/main.js"></script>
```

#### 4. 避免页面中空 href 和 src

当<link>标签的 href 属性为空，或<script>、<img>、<iframe>标签的 src 属性为空时，浏览器在渲染的过程中仍会将 href 属性或 src 属性中的空内容进行加载，直至加载失败，

这样就阻塞了页面中其他资源的下载进程，而且最终加载到的内容是无效的，因此要尽量避免。

```
<!-- 不推荐 -->
<img src="" alt="photo">
<a href="">点击链接</a>
```

## 5. 为 HTML 指定 Cache-Control 或 Expires

为 HTML 内容设置 Cache-Control 或 Expires 可以将 HTML 内容缓存起来，避免频繁向服务器端发送请求。前面讲到，在页面 Cache-Control 或 Expires 头部有效时，浏览器将直接从缓存中读取内容，不向服务器端发送请求。

```
<meta http-equiv="Cache-Control" content="max-age=7200" />
<meta http-equiv="Expires" content="Mon, 20 Jul 2016 23:00:00 GMT" />
```

## 6. 合理设置 Etag 和 Last-Modified

合理设置 Etag 和 Last-Modified 使用浏览器缓存，对于未修改的文件，静态资源服务器会向浏览器端返回 304，让浏览器从缓存中读取文件，减少 Web 资源下载的带宽消耗并降低服务器负载。

```
<meta http-equiv="last-modified" content="Mon, 03 Oct 2016 17:45:57 GMT" />
```

## 7. 减少页面重定向

页面每次重定向都会延长页面内容返回的等待延时，一次重定向大约需要 600 毫秒的时间开销，为了保证用户尽快看到页面内容，要尽量避免页面重定向。

## 8. 使用静态资源分域存放来增加下载并行数

浏览器在同一时刻向同一个域名请求文件的并行下载数是有限的，因此可以利用多个域名的主机来存放不同的静态资源，增大页面加载时资源的并行下载数，缩短页面资源加载的时间。通常根据多个域名来分别存储 JavaScript、CSS 和图片文件。

```
<link rel="stylesheet" href="//cdn1.domain.com/path/main.css">
...
<script src="//cdn2.domain.com/path/main.js"></script>
```

## 9. 使用静态资源 CDN 来存储文件

如果条件允许，可以利用 CDN 网络加快同一个地理区域内重复静态资源文件的响应下载速度，缩短资源请求时间。

## 10. 使用 CDN Combo 下载传输内容

CDN Combo 是在 CDN 服务器端将多个文件请求打包成一个文件的形式来返回的技术, 这样可以实现 HTTP 连接传输的一次性复用, 减少浏览器的 HTTP 请求数, 加快资源下载速度。例如同一个域名 CDN 服务器上的 a.js, b.js, c.js 就可以按如下方式在一个请求中下载。

```
<script src="//cdn.domain.com/path/a.js,b.js,c.js"></script>
```

### 11. 使用可缓存的 AJAX

对于返回内容相同的请求, 没必要每次都直接从服务端拉取, 合理使用 AJAX 缓存能加快 AJAX 响应速度并减轻服务器压力。

```
$.ajax({
  url: url,
  type: 'get',
  cache: true,    // 推荐使用缓存
  data: {}
  success(){
    // ...
  },
  error(){
    // ...
  }
});
```

### 12. 使用 GET 来完成 AJAX 请求

使用 XMLHttpRequest 时, 浏览器中的 POST 方法发送请求首先发送文件头, 然后发送 HTTP 正文数据。而使用 GET 时只发送头部, 所以在拉取服务端数据时使用 GET 请求效率更高。

```
$.ajax({
  url: url,
  type: 'get',    // 推荐使用 get 完成请求
  data: {}
  success(){
    // ...
  },
  error(){
    // ...
  }
});
```

### 13. 减少 Cookie 的大小并进行 Cookie 隔离

HTTP 请求通常默认带上浏览器端的 Cookie 一起发送给服务器, 所以在非必要的情况下, 要尽量减少 Cookie 来减小 HTTP 请求的大小。对于静态资源, 尽量使用不同的域名来存放, 因为 Cookie 默认是不能跨域的, 这样就做到了不同域名下静态资源请求的 Cookie 隔离。

#### 14. 缩小 favicon.ico 并缓存

有利于 favicon.ico 的重复加载，因为一般一个 Web 应用的 favicon.ico 是很少改变的。

#### 15. 推荐使用异步 JavaScript 资源

异步的 JavaScript 资源不会阻塞文档解析，所以允许在浏览器中优先渲染页面，延后加载脚本执行。例如 JavaScript 的引用可以如下设置，也可以使用模块化加载机制来实现。

```
<script src="main.js" defer></script>
<script src="main.js" async></script>
```

使用 async 时，加载和渲染后续文档元素的过程和 main.js 的加载与执行是并行的。使用 defer 时，加载后续文档元素的过程和 main.js 的加载是并行的，但是 main.js 的执行要在页面所有元素解析完成之后才开始执行。

#### 16. 消除阻塞渲染的 CSS 及 JavaScript

对于页面中加载时间过长的 CSS 或 JavaScript 文件，需要进行合理拆分或延后加载，保证关键路径的资源能快速加载完成。

#### 17. 避免使用 CSS import 引用加载 CSS

CSS 中的 @import 可以从另一个样式文件中引入样式，但应该避免这种用法，因为这样会增加 CSS 资源加载的关键路径长度，带有 @import 的 CSS 样式需要在 CSS 文件串行解析到 @import 时才会加载另外的 CSS 文件，大大延后 CSS 渲染完成的时间。

```
<!-- 不推荐 -->
<style>
@import "path/main.css";
</style>

<!-- 推荐 -->
<link rel="stylesheet" href="//cdn1.domain.com/path/main.css">
```

### 👉 页面渲染类

#### 1. 把 CSS 资源引用放到 HTML 文件顶部

一般推荐将所有 CSS 资源尽早指定在 HTML 文档 <head> 中，这样浏览器可以优先下载 CSS 并尽早完成页面渲染。



## 2. JavaScript 资源引用放到 HTML 文件底部

JavaScript 资源放到 HTML 文档底部可以防止 JavaScript 的加载和解析执行对页面渲染造成阻塞。由于 JavaScript 资源默认是解析阻塞的，除非被标记为异步或者通过其他的异步方式加载，否则会阻塞 HTML DOM 解析和 CSS 渲染的过程。

## 3. 不要在 HTML 中直接缩放图片

在 HTML 中直接缩放图片会导致页面内容的重排重绘，此时可能会使页面中的其他操作产生卡顿，因此要尽量减少在页面中直接进行图片缩放。

## 4. 减少 DOM 元素数量和深度

HTML 中标签元素越多，标签的层级越深，浏览器解析 DOM 并绘制到浏览器中所花的时间就越长，所以应尽可能保持 DOM 元素简洁和层级较少。

```
<!-- 不推荐 -->
<div>
  <span>
    <a href="javascript: void(0);">
      
    </a>
  </span>
</div>
```

```
<!-- 推荐 -->

```

## 5. 尽量避免使用<table>、<iframe>等慢元素

<table>内容的渲染是将 table 的 DOM 渲染树全部生成完并一次性绘制到页面上的，所以在长表格渲染时很耗性能，应该尽量避免使用它，可以考虑使用列表元素<ul>代替。尽量使用异步的方式动态添加 iframe，因为 iframe 内资源的下载进程会阻塞父页面静态资源的下载与 CSS 及 HTML DOM 的解析。

## 6. 避免运行耗时的 JavaScript

长时间运行的 JavaScript 会阻塞浏览器构建 DOM 树、DOM 渲染树、渲染页面。所以，任何与页面初次渲染无关的逻辑功能都应该延迟加载执行，这和 JavaScript 资源的异步加载思路是一致的。

## 7. 避免使用 CSS 表达式或 CSS 滤镜

CSS 表达式或 CSS 滤镜的解析渲染速度是比较慢的，在有其他解决方案的情况下应该尽量避免使用。

```
// 不推荐
.opacity{
  filter:progid:DXImageTransform.Microsoft.Alpha(opacity=50);
}
```

### 5.4.3 移动端浏览器前端优化策略

相对于桌面端浏览器，移动端 Web 浏览器上有一些较为明显的特点：设备屏幕较小、新特性兼容性较好、支持一些较新的 HTML5 和 CSS3 特性、需要与 Native 应用交互等。但移动端浏览器可用的 CPU 计算资源和网络资源极为有限，因此要做好移动端 Web 上的优化往往需要做更多的事情。首先，在移动端 Web 的前端页面渲染中，桌面浏览器端上的优化规则同样适用，此外针对移动端也要做一些极致的优化来达到更好的效果。需要注意的是，并不是移动端的优化原则在桌面浏览器端就不适用，而是由于兼容性和差异性的原因，一些优化原则在移动端更具代表性。

#### 👉 网络加载类

##### 1. 首屏数据请求提前，避免 JavaScript 文件加载后才请求数据

为了进一步提升页面加载速度，可以考虑将页面的数据请求尽可能提前，避免在 JavaScript 加载完成后才去请求数据。通常数据请求是页面内容渲染中关键路径最长的部分，而且不能并行，所以如果能将数据请求提前，可以极大程度上缩短页面内容的渲染完成时间。

##### 2. 首屏加载和按需加载，非首屏内容滚屏加载，保证首屏内容最小化

由于移动端网络速度相对较慢，网络资源有限，因此为了尽快完成页面内容的加载，需要保证首屏加载资源最小化，非首屏内容使用滚动的方式异步加载。一般推荐移动端页面首屏数据展示延时最长不超过 3 秒。目前中国联通 3G 的网络速度为 338KB/s (2.71Mb/s)，所以推荐首屏所有资源大小不超过 1014KB，即大约不超过 1MB。

##### 3. 模块化资源并行下载

在移动端资源加载中，尽量保证 JavaScript 资源并行加载，主要指的是模块化 JavaScript 资源的异步加载，例如 AMD 的异步模块，使用并行的加载方式能够缩短多个文件资源的加载时间。

#### 4. inline 首屏必备的 CSS 和 JavaScript

通常为了在 HTML 加载完成时能使浏览器中有基本的样式, 需要将页面渲染时必备的 CSS 和 JavaScript 通过<script>或<style>内联到页面中, 避免页面 HTML 载入完成到页面内容展示这段过程中页面出现空白。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>样例</title>
  <meta name="viewport" content="width=device-width,minimum-scale=1.0,
maximum-scale=1.0,user-scalable=no">
  <style>
    /* 必备的首屏 CSS */
    html, body{
      margin: 0;
      padding: 0;
      background-color: #ccc;
    }
  </style>
</head>
<body>
</body>
</html>
```

#### 5. meta dns prefetch 设置 DNS 预解析

设置文件资源的 DNS 预解析, 让浏览器提前解析获取静态资源的主机 IP, 避免等到请求时才发起 DNS 解析请求。通常在移动端 HTML 中可以采用如下方式完成。

```
<!-- cdn 域名预解析 -->
<meta http-equiv="x-dns-prefetch-control" content="on">
<link rel="dns-prefetch" href="//cdn.domain.com">
```

#### 6. 资源预加载

对于移动端首屏加载后可能会被使用的资源, 需要在首屏完成加载后尽快进行加载, 保证在用户需要浏览时已经加载完成, 这时候如果再去异步请求就显得很慢。

#### 7. 合理利用 MTU 策略

通常情况下, 我们认为 TCP 网络传输的最大传输单元 (Maximum Transmission Unit, MTU) 为 1500B, 即一个 RTT (Round-Trip Time, 网络请求往返时间) 内可以传输的数据量最大为 1500 字节。因此, 在前后端分离的开发模式中, 尽量保证页面的 HTML 内容在 1KB 以内, 这样整

个 HTML 的内容请求就可以在一个 RTT 内请求完成，最大限度地提高 HTML 载入速度。

## 📁 缓存类

### 1. 合理利用浏览器缓存

除了上面说到的使用 Cache-Control、Expires、Etag 和 Last-Modified 来设置 HTTP 缓存外，在移动端还可以使用 localStorage 等来保存 AJAX 返回的数据，或者使用 localStorage 保存 CSS 或 JavaScript 静态资源内容，实现移动端的离线应用，尽可能减少网络请求，保证静态资源内容的快速加载。

### 2. 静态资源离线方案

对于移动端或 Hybrid 应用，可以设置离线文件或离线包机制让静态资源请求从本地读取，加快资源载入速度，并实现离线更新。关于这块内容，我们会在后面的章节中重点讲解。

### 3. 尝试使用 AMP HTML

AMP HTML 可以作为优化前端页面性能的一个解决方案，使用 AMP Component 中的元素来代替原始的页面元素进行直接渲染。

```
<!-- 不推荐 -->
<video width="400" height="300" src="http://www.domain.com/videos/myvideo.mp4"
poster="path/poster.jpg">
  <div fallback>
    <p>Your browser doesn't support HTML5 video</p>
  </div>
  <source type="video/mp4" src="foo.mp4">
  <source type="video/webm" src="foo.webm">
</video>
```

```
<!-- 推荐 -->
<amp-video width="400" height="300" src="http://www.domain.com/videos/myvideo.mp4" poster=
"path/poster.jpg">
  <div fallback>
    <p>Your browser doesn't support HTML5 video</p>
  </div>
  <source type="video/mp4" src="foo.mp4">
  <source type="video/webm" src="foo.webm">
</amp-video>
```

## 📁 图片类

### 1. 图片压缩处理

在移动端，通常要保证页面中一切用到的图片都是经过压缩优化处理的，而不是以原图的形式直接使用的，因为那样很消耗流量，而且加载时间更长。

## 2. 使用较小的图片，合理使用 base64 内嵌图片

在页面使用的背景图片不多且较小的情况下，可以将图片转化成 base64 编码嵌入到 HTML 页面或 CSS 文件中，这样可以减少页面的 HTTP 请求数。需要注意的是，要保证图片较小，一般图片大小超过 2KB 就不推荐使用 base64 嵌入显示了。

```
.class-name {  
    background-image:  
url('data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAoAAAAALCMAAABxsOwqAAAYFBMVEWnxw  
usyQukxQudwQyZvgYhxYfwgyxzAsUHQGOuA0aJAERGAFIXwSTugyEqgtqghghQZgUwQQIpOQKbuguVtQuKrAu  
Cowp2kQlheghTbQZHWQU7SwVAVgQ6TgQlLwMeKwFOemyQAAAAVElEQVQIly3JVraAIAAF0UconXbvf5ei8HfP  
DIQQhBAAFE10iKig3SLRNN4SP/p+N08VC0YnfIlnWtqIkhg/TPYbCvqhqdHAWRXPZSp3g3CWZvVLXC6OJA3ukv  
0AaAAAAAE1FTkSuQmCC');  
}
```

## 3. 使用更高压缩比格式的图片

使用具有较高压缩比格式的图片，如 webp 等。在同等图片画质的情况下，高压压缩比格式的图片体积更小，能够更快完成文件传输，节省网络流量。

```

```

## 4. 图片懒加载

为了保证页面内容的最小化，加速页面的渲染，尽可能节省移动端网络流量，页面中的图片资源推荐使用懒加载实现，在页面滚动时动态载入图片。

```

```

## 5. 使用 Media Query 或 srcset 根据不同屏幕加载不同大小图片

在介绍响应式的章节中我们了解到，针对不同的移动端屏幕尺寸和分辨率，输出不同大小的图片或背景图能保证在用户体验不降低的前提下节省网络流量，加快部分机型的图片加载速度，这在移动端非常值得推荐。

## 6. 使用 iconfont 代替图片图标

在页面中尽可能使用 iconfont 来代替图片图标，这样做的好处有以下几个：使用 iconfont 体积较小，而且是矢量图，因此缩放时不会失真；可以方便地修改图片大小尺寸和呈现颜色。但是需要注意的是，iconfont 引用不同 webfont 格式时的兼容性写法，根据经验推荐尽量按照以

下顺序书写，否则不容易兼容到所有的浏览器上。

```
@font-face {
  font-family: iconfont;
  src: url("../iconfont.eot");
  src: url("../iconfont.eot?#iefix") format("eot"),
       url("../iconfont.woff") format("woff"),
       url("../iconfont.ttf") format("truetype");
}
```

## 7. 定义图片大小限制

加载的单张图片一般建议不超过 30KB，避免大图片加载时间长而阻塞页面其他资源的下载，因此推荐在 10KB 以内。如果用户上传的图片过大，建议设置告警系统，帮助我们观察了解整个网站的图片流量情况，做出进一步的改善。

### 👉 脚本类

#### 1. 尽量使用 id 选择器

选择页面 DOM 元素时尽量使用 id 选择器，因为 id 选择器速度最快。

#### 2. 合理缓存 DOM 对象

对于需要重复使用的 DOM 对象，要优先设置缓存变量，避免每次使用时都要从整个 DOM 树中重新查找。

```
// 不推荐
$('#mod .active').remove('active');
$('#mod .not-active').addClass('active');
```

```
// 推荐
let $mod = $('#mod');
$mod.find('.active').remove('active');
$mod.find('.not-active').addClass('active');
```

#### 3. 页面元素尽量使用事件代理，避免直接事件绑定

使用事件代理可以避免对每个元素都进行绑定，并且可以避免出现内存泄露及需要动态添加元素的事件绑定问题，所以尽量不要直接使用事件绑定。

```
// 不推荐
$('.btn').on('click', function(e){
  console.log(this);
});
```

```
// 推荐
$('body').on('click', '.btn', function(e){
    console.log(this);
});
```

#### 4. 使用 touchstart 代替 click

由于移动端屏幕的设计，touchstart 事件和 click 事件触发时间之间存在 300 毫秒的延时，所以在页面中没有实现 touchmove 滚动处理的情况下，可以使用 touchstart 事件来代替元素的 click 事件，加快页面点击的响应速度，提高用户体验。但同时我们也要注意页面重叠元素 touch 动作的点击穿透问题。

```
// 不推荐
$('body').on('click', '.btn', function(e){
    console.log(this);
});

// 推荐
$('body').on('touchstart', '.btn', function(e){
    console.log(this);
});
```

#### 5. 避免 touchmove、scroll 连续事件处理

需要对 touchmove、scroll 这类可能连续触发回调的事件设置事件节流，例如设置每隔 16ms（60 帧的帧间隔为 16.7ms，因此可以合理地设置为 16ms）才进行一次事件处理，避免频繁的事件调用导致移动端页面卡顿。

```
// 不推荐
$('.scroller').on('touchmove', '.btn', function(e){
    console.log(this);
});

// 推荐
$('.scroller').on('touchmove', '.btn', function(e){
    let self = this;
    setTimeout(function(){
        console.log(self);
    }, 16);
});
```

#### 6. 避免使用 eval、with，使用 join 代替连接符+，推荐使用 ECMAScript 6 的字符串模板

这些都是一些基础的安全脚本编写问题，尽可能使用较高效率的特性来完成这些操作，避免不规范或不安全的写法。

## 7. 尽量使用 ECMAScript 6+的特性来编程

ECMAScript 6+一定程度上更加安全高效, 而且部分特性执行速度更快, 也是未来规范的需要, 所以推荐使用 ECMAScript 6+的新特性来完成后面的开发。

### 渲染类

#### 1. 使用 Viewport 固定屏幕渲染, 可以加速页面渲染内容

一般认为, 在移动端设置 Viewport 可以加速页面的渲染, 同时可以避免缩放导致页面重排重绘。在移动端固定 Viewport 设置的方法如下。

```
<!-- 设置 viewport 不缩放 -->
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
user-scalable=no">
```

#### 2. 避免各种形式重排重绘

页面的重排重绘很耗性能, 所以一定要尽可能减少页面的重排重绘, 例如页面图片大小变化、元素位置变化等这些情况都会导致重排重绘。

#### 3. 使用 CSS3 动画, 开启 GPU 加速

使用 CSS3 动画时可以设置 `transform: translateZ(0)` 来开启移动设备浏览器的 GPU 图形处理加速, 让动画过程更加流畅。

```
-webkit-transform: translateZ(0);
-ms-transform: translateZ(0);
-o-transform: translateZ(0);
transform: translateZ(0);
```

#### 4. 合理使用 Canvas 和 requestAnimationFrame

选择 Canvas 或 requestAnimationFrame 等更高效的动画实现方式, 尽量避免使用 setTimeout、setInterval 等方式来直接处理连续动画。

#### 5. SVG 代替图片

部分情况下可以考虑使用 SVG 代替图片实现动画, 因为使用 SVG 格式内容更小, 而且 SVG DOM 结构方便调整。

#### 6. 不滥用 float

在 DOM 渲染树生成后的布局渲染阶段, 使用 float 的元素布局计算比较耗性能, 所以尽量



减少 float 的使用，推荐使用固定布局或 flex-box 弹性布局的方式来实现页面元素布局。

#### 7. 不滥用 web 字体或过多 font-size 声明

过多的 font-size 声明会增加字体的大小计算，而且也没有必要的。

### 📁 架构协议类

#### 1. 尝试使用 SPDY 和 HTTP 2

在条件允许的情况下可以考虑使用 SPDY 协议来进行文件资源传输，利用连接复用加快传输过程，缩短资源加载时间。HTTP 2 在未来也是可以考虑尝试的。

#### 2. 使用后端数据渲染

使用后端数据渲染的方式可以加快页面内容的渲染展示，避免空白页面的出现，同时可以解决移动端页面 SEO 的问题。如果条件允许，后端数据渲染是一个很不错的实践思路。后面的章节会详细介绍后端数据渲染的相关内容。

#### 3. 使用 Native View 代替 DOM 的性能劣势

可以尝试使用 Native View 的 MNV\* 开发模式来避免 HTML DOM 性能慢的问题，目前使用 MNV\* 的开发模式已经可以将页面内容渲染体验做到接近客户端 Native 应用的体验了。

关于页面优化的常用技术手段和思路主要包括以上这些，尽管列举出很多，但仍可能有少数遗漏，可见前端性能优化不是一件简简单单的事情，其涉及的内容很多。大家可以根据实际情况将这些方法应用到自己的项目当中，要想全部做到几乎是不可能的，但做到用户可接受的原则还是很容易实现的。

世界上没有十全十美的事情，在我们做到了极致优化的同时也付出了很大的代价，这也是前端优化的一个问题。理论上这些优化都是可以实现的，但是作为工程师我们也要明白懂得权衡。优化提升了用户体验，使数据加载更快，但是项目代码却可能打乱，异步内容要拆分出来，首屏的一个雪碧图可能要分成两个，页面项目代码维护成本成倍增加，项目结构也可能变得混乱。所以前期在设计构建、组件的解决方案时要解决好异步的自动处理问题。任何一部分优化都可以做得很深入，但不一定都值得，在优化的同时也要尽量考虑性价比，这才是我们作为一名前端工程师处理前端优化时应该具有的正确思维。

## 5.5 前端用户数据分析

在现代互联网产品的开发迭代中,对前端用户数据的统计分析严重影响着最终产品的成败。谈到前端数据,涉及的方面就比较广了。网站用户数据统计分析通常可以反映出网站的用户规模、用户使用习惯、用户的内容偏好等,了解了这些就能帮助我们调整产品策略、改进产品需求、提高产品质量,除此之外用户数据的统计甚至也会直接和广告收入相关联,那么这一节我们就来总结一下与前端用户数据相关的内容。

### 5.5.1 用户访问统计

先来看看用户的访问统计,通常页面上用户访问统计主要包括 PV (Page View)、UV (Unique Visitor)、VV (Visit View)、IP (访问站点的不同 IP 数) 等。PV 一般指在一天时间之内页面被所有用户访问的总次数,即每一次页面刷新都会增加一次 PV; UV 是指在一天时间之访问内页的不同用户个数,和 PV 不同的是,如果一个页面在同一天内被某个相同用户多次访问,只计算一次 UV; VV 是统计网站被用户访问次数的参考数据,通常用户从进入网站到最终离开该网站的整个过程只算一次 VV; IP 则表示一天时间之内访问网站的不重复 IP 数,一天时间内相同 IP 地址多次访问同一个网站只计算一次。

#### 👉 PV

PV 作为单个页面的统计量参数,通常用来统计获取关键入口页面或临时推广性页面的访问量或推广效果,由于 PV 的统计一般是不做任何条件限制的,可以人为地刷新来提升统计量,所以单纯靠 PV 是无法反应页面被用户访问的具体情况的。

#### 👉 UV

UV 则可以认为是前端页面统计中一个最有价值的统计指标,因为其直接反应页面的访问用户数。目前有较多站点的 UV 是按照一天之内访问目标页面的 IP 数来计算的,因此我们也可以根据 UV 来统计站点的周活跃用户量和月活跃用户量。严格来讲,根据一天时间内访问目标页面的 IP 数来计算 UV 是不严谨的,因为在办公区或校园局域网的情况下,多个用户访问互联网网站的 IP 可能是同一个,但实际上的访问用户却有很多。所以为了得到更加准确的结果,除了根据 IP,还需要结合其他的辅助信息来识别统计不同用户的 UV,这里介绍两种常用的方式。

- 根据浏览器 Cookie 和 IP 统计。在目标页面每次打开时向浏览器中写入唯一的某个 Cookie 信息,再结合 IP 一起上报统计,就可以精确统计出一天时间内访问页面的用户

数。存在的问题是，如果用户手动清除了 Cookie 再进入访问，页面被重新访问时就只能算第二次。

- 结合用户浏览器标识 `userAgent` 和 IP 统计。由于使用 Cookie 统计存在可能被手动清除的问题，所以推荐结合浏览器标识 `userAgent` 来统计。这样可以在一定程度上区分同 IP 下的不同用户，但也不完全准确，IP 和浏览器标识 `userAgent` 相同的情况也很常见，但仍却只能计算一次。

由此可见，虽然 UV 是网站统计的一个很重要的统计量，但一般情况下是无法用于精确统计的，所以通常需要结合 PV、UV 来一起分析网站被用户访问的情况。此外，我们还可以对站点一天的新访客数、新访客比率等进行统计，计算第一次访问网站的新用户数和比例，这对判断网站用户增长也是很有意义的。

#### 👉 VV

PV 和 UV 更多是针对单页面进行的统计，而 VV 则是用户访问整个网站的统计指标。例如用户打开站点，并在内部做了多次跳转操作，最后关闭该网站所有的页面，即为一次 VV。

#### 👉 IP

IP 是一天时间内访问网页或网站的独立 IP 数，一般服务器端可以直接获取用户访问网站时的独立 IP，统计也比较容易处理。需要注意的是，我们要理解 IP 统计与 UV 统计的区别和联系。

## 5.5.2 用户行为分析

对于较小的项目团队来说，或许得到页面或网站的 PV、UV、VV、IP 这些基本的统计数据就可以了。其实不然，相对于访问量的统计，用户行为分析才是更加直接反映网页内容是否受用户喜欢或满足用户需求的一个重要标准，用户在页面上操作的行为有很多种，每种操作都可能对应页面上不同的展示内容。如果我们能知道用户浏览目标页面时所有的行为操作，一定程度上就可以知道用户对页面的哪些内容感兴趣，对哪些内容不感兴趣，这对产品内容的调整和改进是很有意义的。一般用于分析用户行为的参数指标主要包括：页面点击量、用户点击流、用户访问路径、用户点击热力图、用户转换率、用户访问时长分析和用户访问内容分析等，下面我们来逐一分析。

#### 👉 页面点击量

页面点击量用来统计用户对于页面某个可点击或可操作区域的点击或操作次数。以点击的

情况为例，统计页面上某个按钮被点击的次数就可以通过该方法来计算，这样通过统计的结果可以分析出页面上哪些按钮对应的内容是用户可能感兴趣的。

### 👉 用户点击流分析

点击流用来统计用户在页面中发生点击或操作动作的顺序，可以反映用户在页面上的操作行为。所以统计上报时需要在浏览器上先保存记录用户的操作顺序，例如在关键的按钮中埋点，点击时向 `localStorage` 中记录点击或操作行为的唯一 `id`，在用户一次 `VV` 结束或在下一次 `VV` 开始时进行点击流上报，然后通过后台归并统计分析。

### 👉 用户访问路径分析

用户访问路径和用户点击流有点类似，不过用户访问路径不针对用户的可点击或操作区域埋点，而是针对每个页面埋点记录用户访问不同页面的路径。上报信息的方法和用户点击流上报相同，常常也是在一次 `VV` 结束或下一次 `VV` 开始时上报用户的访问路径。

以图 5-6 所示的某电影票的购买流程为例，页面首页、列表页、详情页、影评页、购票页、购票完成页对应的埋点标识分别为 A、B、C、D、E、F，某用户进入首页后选择最近上映的电影列表，查看某个电影介绍和影评，然后选择购票，购票成功后又返回首页并退出，那么我们可以记录该用户这次 `VV` 的访问路径为 A-B-C-D-E-F-A，将该路径存入 `localStorage` 中，在 `VV` 结束前或下一次 `VV` 开始时将用户访问路径字符串上报到服务器。再结合点击流路径，就可以基本分析出用户这次访问过程中所有点击的操作和访问的页面路径了。

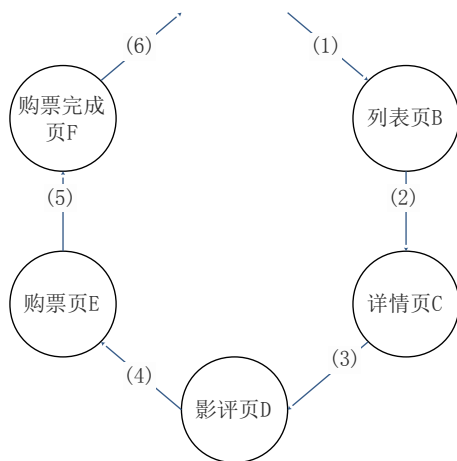


图 5-6 用户点击购票流程

## 👉 用户点击热力图

用户点击热力图是为了统计用户的点击或操作发生在整个页面哪些区域位置的一种分析方法,一般是统计用户操作习惯和页面某些区域内容是否受用户关注的一种方式。图 5-7 为 google 某个关键词的用户点击搜索结果热力图,从图中可以分析出,搜索结果中左上方的结果更容易被用户点击,所以这些地方的搜索结果通常是更具有价值的。

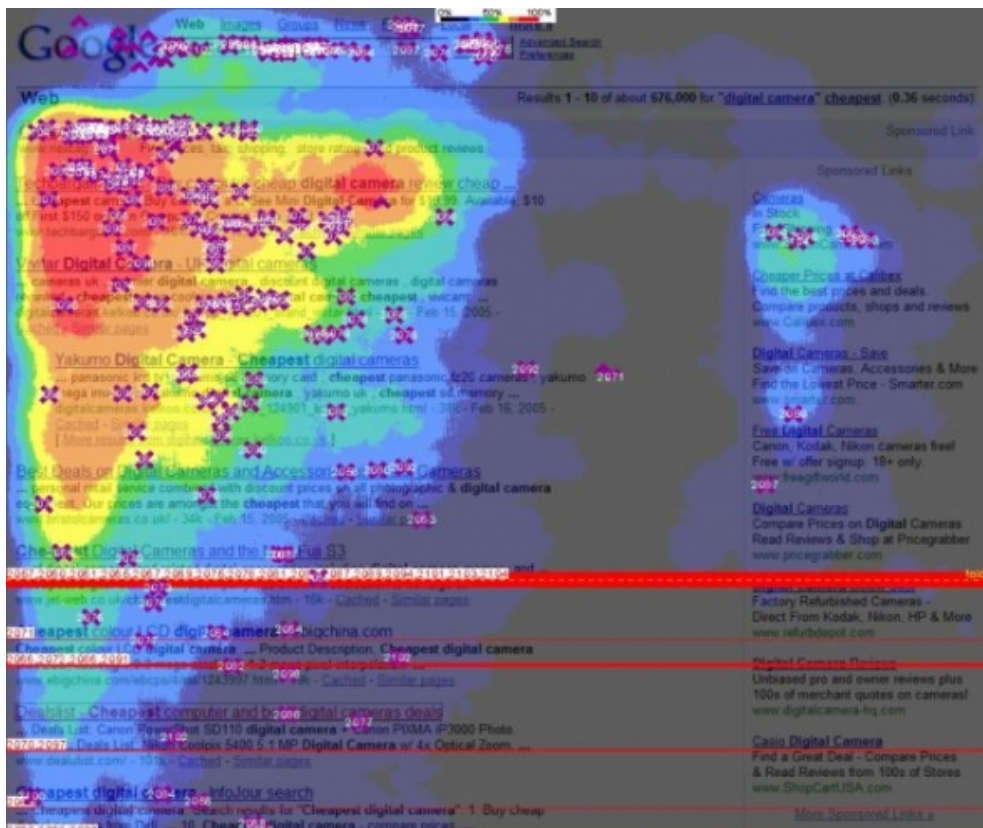


图 5-7 页面点击热力图示例

这种统计方法获取上报点的方式主要是捕获鼠标事件在屏幕中的坐标位置进行上报,然后在服务端进行计算归类分析并绘图。通过如下代码可以简单地获取点击事件在页面中的位置并进行上报。

```
$(document).on('click', 'body', function(e){
    report(e.pageX, e.pageY); // e.pageX 和 e.pageY 为相对于整个页面的坐标
});
```

### 👉 用户转化率与导流转化率

对用户转化率的分析一般在一些临时推广页面或拉取新用户宣传页面上比较常用，这里统计也很简单，例如要统计某个新产品推广页面的用户转化率，通过计算经过该页面注册的用户数相对于页面的 PV 比例就可以得出。

用户转化率 = 通过该页面注册的用户数/页面 PV

相对来说，用户转化率分析的应用场景比较单一。还有另一种导流的页面统计分析和该页面的功能类似，不过其作用是将某个页面的用户访问流量引导到另一个页面中，导流转化率可以用通过源页面导入的页面访问 PV 相对于源页面的总 PV 比例来表示。

导流转化率 = 通过源页面导入的页面访问 PV/源页面 PV

本质上，关键的统计分析仍是对现有页面访问量进行对比和计算而得出的，并不是统计出来的。

### 👉 用户访问时长、内容分析

用户访问时长和内容分析则是统计分析用户在某些关键内容页面的停留时间，来判断用户对该页面的内容是否感兴趣，从而分析出用户对网站可能感兴趣的内容，方便以后精确地向该用户推荐他们感兴趣的内容。

## 5.5.3 前端日志上报

接触过后端开发的人可能知道，一般在程序运行出现异常时可以通过写服务器日志的方式来记录错误的信息，然后下载服务器日志打开查看是哪里的的问题并进行修复。看似完美，但是如果是前端页面运行出现了问题，我们却不能打开用户浏览器的控制台记录来查看代码中到底出现了什么错误。

一般情况下，在前端开发中，前端工程师按照需求完成页面开发，通过产品体验确认和测试，页面就可以上线了。但不幸的是，产品很快就收到了用户的投诉。用户反映页面点击按钮没反应而且能复现，我们试了一下却一切正常，于是追问用户所用的环境，最后结论是用户使用了一个非常小众的浏览器打开页面，因为该浏览器不支持某个特性，因此页面报错，整个页面停止响应。在这种情况下，用户反馈的投诉花掉了我们很多时间去定位问题，然而这并不是最可怕的，更让我们担忧的是更多的用户遇到这种场景后便会直接抛弃这个有问题的“垃圾产品”。这个问题唯一的解决办法就是在尽量少的用户遇到这样的场景时就把问题即时修复掉，保

证尽量多的用户可以正常使用。首先我们需要在少数用户使用产品出错时知道有用户出错，而且尽量定位到是什么错误。由于用户的运行环境是在浏览器端的，因此可以在前端页面脚本执行出错时将错误信息上传到服务器，然后打开服务器收集的错误信息进行分析来改进产品的质量。要实现这个过程，我们必须考虑下面几个问题。

### 👉 怎样获取错误日志

浏览器提供了 `try...catch` 和 `window.onerror` 的两种机制来帮助我们获取用户页面的脚本错误信息。

一般来说，使用 `try...catch` 可以捕捉前端 JavaScript 的运行时错误，同时拿到出错的信息，例如错误信息描述、堆栈、行号、列号、具体的出错文件信息等。我们也可以在这个阶段将用户浏览器信息等静态内容一起记录下来，快速地定位问题发生的原因。需要注意的是，`try...catch` 无法捕捉到语法错误，只能在单一的作用域内有效捕获错误信息，如果是异步函数里面的内容，就需要把 `function` 函数块内容全部加入到 `try...catch` 中执行。

```
try{
    // 单一作用域 try...catch 可以捕获错误信息并进行处理
    console.log(obj);
}catch(e){
    console.log(e); //处理异常, ReferenceError: obj is not defined
}

try{
    // 不同作用域不能捕获到错误信息
    setTimeout(function() {
        console.log(obj); // 直接报错, 不经过 catch 处理
    }, 200);
}catch(e){
    console.log(e);
}

// 同一个作用域下能捕获到错误信息
setTimeout(function() {
    try{
        // 当前作用域 try...catch 可以捕获错误信息并进行处理
        console.log(obj);
    }catch(e){
        console.log(e); //处理异常, ReferenceError: obj is not defined
    }
}, 200);
```

在上面的这个例子中，`try...catch` 无法获取异步函数 `setTimeout` 或其他作用域中的

错误信息，这样就只能在每个函数里面添加 `try...catch` 了。相比之下，`window.onerror` 的方法可以在任何执行上下文中执行，如果给 `window` 对象增加一个错误处理函数，便既能处理捕获错误又能保持代码的优雅性了。`window.onerror` 一般用于捕捉脚本语法错误和运行时错误，可以获得出错的文件信息，如出错信息、出错文件、行号等，当前页面执行的所有 JavaScript 脚本出错都会被捕捉到。

```
window.onerror = function (msg, url, line){
    // 可以捕获异步函数中的错误信息并进行处理，提示 Script error.
    console.log(msg); // 获取错误信息
    console.log(url); // 获取出错的文件路径
    console.log(line); // 获取错误出错的行数
};

setTimeout(function() {
    console.log(obj); // 可以被捕捉到，并在 onerror 中处理
}, 200);
```

然而，使用 `onerror` 要注意，在不同浏览器中实现函数处理返回的异常对象是不相同的，而且如果报错的 JavaScript 和 HTML 不在同一个域名下，错误时 `window.onerror` 中的 `errorMsg` 全部为 `script error` 而不是具体的错误描述信息，此时需要添加 JavaScript 脚本的跨域设置。

```
<script src="//www.domain.com/main.js" crossorigin></script>
```

如果服务器因为一些原因不能设置跨域或设置起来比较麻烦，那就只能在每个引用的文件里添加 `try...catch` 进行处理。

虽然使用 `window.onerror` 可以获取页面的出错信息、出错文件和行号，但是 `window.onerror` 有跨域限制，如果需要获取错误发生的具体描述、堆栈内容、行号、列号和具体的出错文件等详细日志，就必须使用 `try...catch`，但是 `try...catch` 又不能在多个作用域中统一处理错误。

幸运的是，我们可以对前端脚本中常用的异步方法入口函数或模块引用的入口方法统一使用 `try...catch` 进行一层封装，这样就可以使用 `try...catch` 捕获每个引用模块作用域下的主要错误信息了。例如我们就可以对 `setTimeout` 函数用如下方式进行封装并捕获错误信息。

```
// 对 setTimeout 实现函数进行包装
window.setTimeoutTry = function(fn, time) {
    let args = arguments;
    let _fn = function() {
        try {
            return fn.apply(this, args); // 将函数参数用 try...catch 包裹
        } catch (e) {
            console.log(e);
        }
    };
    setTimeout(_fn, time);
};
```



```
    }  
  }  
  return window['setTimeout'](_fn, time);  
}  
  
try {  
  setTimeoutTry(function() {  
    obj // 这获取错误信息, ReferenceError: obj is not defined  
  }, 300);  
} catch (e) {  
  console.log(e);  
}
```

我们可以对不同作用域的 `setTimeout` 参数函数的引入方式使用 `try...catch` 进行封装, 让 `try...catch` 能捕获到 `setTimeout` 脚本中的错误并使用 `setTimeoutTry` 函数来代替。对于异步引入模块定义函数 `require` 或 `define` 也可以进行类似的封装, 这样就可以获取到不同模块里面作用域的错误信息了。因此, 这里捕获错误的方式可以根据具体的条件和场景灵活选择, 在没有特别限制的情况下, 使用 `window.onerror` 是比较高效、便捷的。

### 📌 怎样将错误信息上传到服务器

如果捕获到了具体的错误或栈信息, 就可以将错误信息进行上报了, 如出错信息、错误行号、列号、用户浏览器信息等, 通过创建 **HTTP** 请求的方式即可将它们发送到日志收集服务器, 这些对于迅速定位和解决问题是大有裨益的。当然错误信息上报设计时需要注意一点: 页面的访问量可能很大, 如果到达百万级、千万级, 那么就需要按照一定的条件上报, 例如根据一定的概率进行上报, 否则大量的错误信息上报请求会占用日志收集服务器的很多资源和流量。

### 📌 怎样通过高效的方式来找到问题

为了方便查看收集到的这些信息, 我们通常可以建立一个简单的内容管理系统 (Content Management System, CMS) 来管理查看错误日志, 对同一类型的错误做归并统计, 也可以建立错误量实时统计来查看错误量的即时变化情况。当某个版本发布后, 如果收到的错误量明显增加, 就需要格外注意。另外一点要注意的是, 上报错误信息机制是用来辅助产品质量改进的, 不能因为在页面中添加了错误信息收集和上报而影响了原有的业务模块功能。

### 📌 文件加载失败监控

如果要进一步完善地检测页面的异常信息, 可以尝试对静态资源文件加载失败的情况进行监控。例如在 **CDN** 网络中, 可能因为部分机器故障, 导致用户加载不到 `<img>`、`<script>` 等静态资源, 但是开发者不一定能复现, 而且无法第一时间知道静态资源加载失败了。这种情况下这就需要在页面上自动捕获文件加载失败的异常来进行处理, 可以对 `<img>` 或 `<script>`

标签元素的 `readyChange` 进行是否加载成功的判断。不幸的是，只有部分 IE 浏览器支持 `<img>` 或 `<script>` 的 `readyState`，因此一般还需要结合其他方式，如 `onload`，针对不同浏览器分开处理。

通过这种方法仅仅是判断了文件或脚本加载成功的情况，我们还需要指定一个文件加载的列表，在一段时间后将页面文件加载的结果对象上报给服务器端来统计不同文件的具体加载情况。

```
// 页面需要加载的三个 script 脚本资源
let scripts = [script1, script2, script3];

// 三个 script 加载的初始状态
let loaded = {
  [script1]: false,
  [script2]: false,
  [script3]: false
}

for(let script of scripts){
  // IE 浏览器的情况设置 readyState 来判断
  if (script.readyState) {
    script.onreadystatechange = function() {
      let state = this.readyState;
      if (state === 'loaded' || state === 'complete') {
        callback(); // 脚本加载成功回调
        // 表示该脚本加载成功
        loaded[script] = true;
      }
    }
  } else {
    // 其他浏览器，如 Firefox、Safari、Chrome 或 Opera，结合 onLoad
    script.onload = function() {
      callback(); // 脚本加载成功回调
      // 表示该脚本加载成功
      loaded[script] = true;
    }
  }
}

setTimeout(function(){
  // 例如 15 秒后执行页面脚本加载情况的上报进行统计
  report(loaded);
}, 15000);
```

在上面的例子中，我们在这三个脚本加载成功时将文件加载是否成功的标志位改为 `true`，一段时间后进行上报，如果某个文件加载没有成功，其地址将会上报到服务器端并被我们收集

到，然后通过分析和统计就可以监控到文件加载失败的情况了。

### 5.5.4 前端性能分析上报

在本章第四节中，我们讲到了前端性能测试方法和优化方法。需要注意的是，开发者怎样知道用户端打开页面时的性能如何呢，一个可行的方法就是将页面性能数据进行上报统计，例如将 **Performance Timing** 数据、开发者自己埋点的性能统计数据通过页面 **JavaScript** 统一上报到远程服务器，在服务器端统计计算性能数据的平均值来评判前端具体页面的性能情况。移动端首屏建议加载的最长时间为 3 秒，推荐为 1.5 秒以内。PC 端推荐为 1 秒以内，最长不超过 1.5 秒。如果检测到页面首屏加载的时间超出了推荐的最大值时，那就需要使用前端性能优化手段进行优化了。

以上介绍的是前端页面数据统计和分析的主要内容，在实际项目中我们可以根据产品或开发需要来进行实践。需要注意的是，不要过度设计，例如对于访问量很少的网站进行大量的用户行为分析可能就得得不偿失了。

## 5.6 前端搜索引擎优化基础

搜索引擎优化简称 **SEO**，知道的人也许很多，但关注的人并不是太多，因为我们常常是在已有的项目上面进行开发，或者进行 **Hybrid** 页面开发，遇到的场景并不多。作为前端工程师，了解搜索引擎优化方面的相关知识是很重要的，这一节我们来一起学习关于搜索引擎优化方面需要注意的内容。

### 5.6.1 title、keywords、description 的优化

**title**、**keywords**、**description** 是可以在 **HTML** 的 `<meta>` 标签内定义的，有助于搜索引擎抓取到网页的内容。要注意的是，一般 **title** 的权重是最高的，也是最重要的，因此我们应该好好利用 **title** 来提高页面的权重。**keywords** 相对权重较低，可以作为页面的辅助关键词搜索。**description** 的描述一般会直接显示在搜索结果的介绍中，可以使用户快速了解页面内容的描述文字，所以要尽量让这段文字能够描述整个页面的内容，增加用户进入页面的概率。

#### title 的优化

一般 **title** 的设置要尽量能够概括页面的内容，可以使用多个 **title** 关键字组合的形式，并用分隔符连接起来。分隔符一般有 “\_”、“-”、“ ”、“,” 等，其中 “\_” 分隔符比较容易被百度搜

索引引擎检索到，“-”分隔符则容易被谷歌搜索引擎检索到，“,”则在英文站点中使用比较多，可以使用空格。**title** 的长度在桌面浏览器端一般建议控制在 30 个字以内，在移动端控制在 20 个字以内，若长度超出时浏览器会默认截断并显示省略号。在实际开发中，因为具体业务的关系，我们可能更多针对百度搜索引擎进行优化，下面提出几点关于百度搜索引擎优化的建议。

关于 **title** 格式的优化设置可以遵循以下规则。

- 每个网页都应该有独一无二的标题，切忌所有的页面都使用同样的默认标题
- 标题主题明确，应该包含网页中最重要的信息
- 简明精练，不应该罗列与网页内容不相关的信息
- 用户浏览通常从左到右的，建议将重要的内容放到 **title** 靠前的位置
- 使用用户所熟知的语言描述，如果有中、英文两种网站名称，尽量使用用户熟知的语言作为标题描述

对于网站不同页面 **title** 的定义可以设置如下。

- 首页：网站名称\_提供服务介绍或产品介绍
- 列表页：列表名称\_网站名称
- 文章页：文章标题\_文章分类\_网站名称
- 如果文章标题不是很长，还可以增加部分关键词来提高网页的检索量，如文章 **title\_关键词\_网站名称**

例如某个博客的名称为极限前端，那么其首页的 **title** 就可以如下编写。

```
<!-- 不好的 title 设置 -->
<title>极限前端</title>
<title>极限前端_front end</title>

<!-- 良好的 title 设置 -->
<title>极限前端_首页_前端技术知识_某某某的博客</title>
```

## 👉 keywords

**keywords** 是目前用于页面内容检索的辅助关键字信息，容易被搜索引擎检索到，所以恰当的设置页面 **keywords** 内容对于页面的 **SEO** 也是很重要的，而且 **keywords** 本身的使用也比较简单。

### 👉 description 优化

在搜索引擎检索结果中，`description` 更重要的作用是作为搜索结果的描述，而不是作为权重计算的重要参考因素。`description` 的长度在桌面浏览器页面中一般为 78 个中文字符，移动端为 50 个，超过则会自动截断并显示省略号。如下定义 `title`、`keywords`、`description` 比较合适。

```
<!-- 不好的 title、keywords、description 优化设置 -->
<title>极限前端</title>
<meta name="keywords" content="极限前端">
<meta name="description" content="极限前端">

<!-- 良好的 title、keywords、description 优化设置 -->
<title>前端搜索引擎优化基础_极限前端_前端技术知识_某某某的博客</title>
<meta name="keywords" content="现代前端技术，前端页面 SEO 优化，极限前端，某某某的博客">
<meta name="description" content="本章讲述了前端搜索引擎优化基础实践技术。">
```

## 5.6.2 语义化标签的优化

`title`、`keywords`、`description` 的设置对页面 SEO 具有重要意义，但除了页面 `title`、`keywords`、`description` 外，还有我们通常说的页面结构语义化设计，因为搜索引擎分析页面内容时可以解析语义化的标签来获取内容，并赋予相关的权重，因此语义化结构的页面就比全部为 `<div>` 标签元素布局的页面更容易被检索到。结合语义化标签我们可以实现更多的功能。

### 👉 使用具有语义化的 HTML5 标签结构

如果页面兼容性条件允许（主要是指浏览器兼容性允许），尽量使用 HTML5 语义化结构标签。如图 5-8 所示，使用 `<header>`、`<nav>`、`<aside>`、`<article>`、`<footer>` 等标签增加页面的语义化内容，可以让搜索引擎更容易获取页面的结构内容。

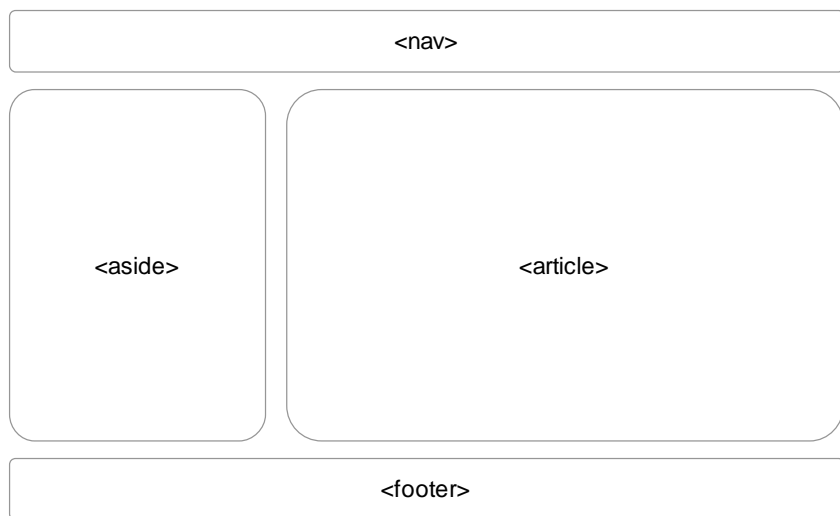


图 5-8 SEO 友好的页面结构

### 👉 唯一的H1 标题

建议每个页面都有一个唯一的<h1>标题，但一般<h1>内容并不是网站的标题。<h1>作为页面最高层级的标题能够更容易被搜索引擎收录，并赋予页面相对较高权重的内容描述。一般我们设置首页的<h1>标题为站点名称，其他内页的<h1>标题则可以为各个内页的标题，如分类页用分类的名字、详情页用详情页标题等。

因为 SEO 的需要，应该尽量保证搜索引擎抓取到的页面是有内容的，但是以 AJAX 技术实现的 SPA 应用在 SEO 上不具有优势，因此要尽量避免这样的页面实现方式，后面会讲解如何使用后端数据渲染的方式来解决 AJAX 类型网站的 SEO 问题。

### 👉 <img>添加alt属性

一般要求<img>标签必须设置 alt 属性，这样更有利于搜索引擎检索出图片的描述信息。

## 5.6.3 URL规范化

统一网站的地址链接：

`http://www.domain.com`  
`http://domain.com`

```
http://www.domain.com/index.html  
http://domain.com/index.html
```

以上四个地址都可以表示跳转到同一个站点的首页，虽然不会对用户访问造成什么麻烦，但对于搜索引擎来说是四条网址并且内容相同。这种情况有可能会被搜索引擎误认为是作弊手段，另外当搜索引擎要规范化网址时，需要从这些选择中挑一个作为代表，但是挑的这一个不一定是最好的，因此我们最好统一搜索引擎访问页面的地址，否则可能影响网站入口搜索结果的权重。

### 👉 301 跳转

如果 URL 发生改变，一定要使旧的地址 301 指向新的页面，否则搜索引擎会把原有的这个 URL 当作死链处理，之前完成的页面内容收录权重的工作就都失效了。

### 👉 canonical

当该页面有不同参数传递的时候，标签属性也可以起到标识页面唯一性的作用，例如以下三个地址。

```
//:domain.com/index.html  
//:domain.com/index.html?from=123  
//:domain.com/index.html?from=456
```

在搜索引擎中，以上三个地址分别表示三个页面，但其实后面两个一般表示页面跳转的来源，所以为了确保这三个地址为同一个页面，往往在<head>上加上 canonical 声明，告诉搜索引擎在收录页面时可以按照这个 href 提供的页面地址去处理，而不是将每个地址都独立处理。

```
<link rel="canonical" href="//:domain.com/index.html" />
```

## 5.6.4 robots

robots.txt 是网站站点用来配置搜索引擎抓取站点内容路径的一种控制方式，放置于站点根目录下。搜索引擎爬虫访问网站时会访问 robots.txt 文件，robots.txt 可以指导搜索引擎爬虫禁止抓取网站某些内容或只允许抓取哪些内容，这就保证了搜索引擎不抓取站点中临时或不重要的内容，保证网站的主要内容被搜索引擎收录。

## 5.6.5 sitemap

sitemap 格式一般分为 HTML 和 XML 两种，命名可以为 sitemap.html 或 sitemap.xml，作用是列出网站所有的 URL 地址，方便搜索引擎去逐个抓取网站的页面，增加网站页面在搜索

引擎中的曝光量。

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>//www.domain.com</loc>
    <lastmod>2016-12-28</lastmod>
    <changefreq>always</changefreq>
    <priority>1.0</priority>
  </url>
  ...
</urlset>
```

如上述代码所示，其中<urlset>、<url>、<loc>为必须标签，<lastmod>、<changefreq>、<priority>为可选标签，<lastmod>表示页面最后一次的更新时间，<changefreq>表示文件更新频率，<priority>表示 URL 相对的重要程度，取值范围为 0~1.0，1.0 表示最重要，一般用在首页上。这样搜索引擎收录网站时不仅可以遍历站点所有地址，还可以给予不同页面不同的权重。

关于 SEO 的内容有很多，本节只是用简短的篇幅讲解了实际开发中可能涉及的部分。如果要深入研究 SEO，可以读一本专门针对于搜索引擎优化方面的书（一般都是很厚的一本）。

## 5.7 前端协作

前端技术从来都不是独立存在的，它是互联网应用复杂到一定程度后衍生出来的一块分支技术领域。前端技术涉及 UI 界面、数据展示、用户交互等实现，因此作为一名合格的前端工程师就不可避免地要和团队其他成员进行协作沟通，如产品经理、UI 设计师、交互设计师、后台工程师、运维工程师等。这一节我们就来聊聊前端工程师应该如何高效地完成团队协作的使命。

### 5.7.1 沟通能力和沟通技巧

可能是因为工作性质的缘故，工程师常常比较内向。但是对于团队合作的项目，沟通必不可少，团队协作中的沟通意义重大。

什么是沟通能力？通俗地说就是你向别人传达信息或从别人那里获取信息的能力。沟通技巧就是为了高效传达信息或获取信息使用的方法和手段。例如，产品经理需求宣讲时有需求 PPT、描述、交互图等，那么需求就是要传递的信息，会议、PPT、描述、交互图等是方法和手段，即沟通技巧。学会了高效的沟通技巧可以大大减少沟通所消耗的时间，提升整体工作效率。



相反地，工程师有时也会成为信息的传递者。例如，你向产品经理提出他的需求问题或改进建议时，对方如果能很快明白你要讲的内容，说明这次沟通是高效的。前端工程师需要合作或沟通的角色多种多样，所以我们平时要注意提高沟通效率来节省沟通时间，从而提升工作效率。

### 5.7.2 与产品经理的“对抗”

关于工程师和产品经理有无数经典幽默的段子，但是我要说的不是这些。产品经理通常是直接给我们提出开发需求的团队角色，这是他的工作。关于产品经理，推荐前端工程师去看一两本关于该角色的书籍来了解一下这个工作，这样才能知己知彼，共同提高效率。

对于产品经理的需求，我们要完成，但同时务必要考虑以下几点。

- 产品经理提出的需求是否明确。要先弄明白需求，如果需求不明确，一定要找提需求的人确认清楚，不要盲目写代码。
- 技术方案是否可行，即需求开发的难度评估。一般产品经理提需求之前会评估实现的难度，但不一定准确，如果某个功能实现难度较大超出了产品经理的预期，一定要提出来，否则后期容易出现风险。
- 需求性价比是否够高。如果某个功能点实现代价很大，但是功能性价值一般，就要和产品经理沟通，是否一定要做。曾经遇到过极少数新人产品经理提出需要翻新架构才能完成的需求，这种情况确认不可行后建议直接驳回，但这种情况极少。
- 需求是否合理。产品经理很多时候设计功能是基于客观分析的主观设计，不可能做到完全正确，有时提出的需求都不太合理，那就要对他提的需求多质疑几次，在开始之前把这些问题过滤掉。
- 风险管理。产品需求常常会变更，开发也会延期，如果因为需求变更导致延期一定要告知产品经理，询问其是否接受，不要等到需求交付时再告诉大家并未完成。

每个人遇到的情况可能不一样，但有一条相同——尽量想的更多一些，理解和尊重合作伙伴，因为能一起合作都是彼此的荣幸。

### 5.7.3 与后台工程师的合作

另一个和前端工程师站在同一战线的角色可能就是后台开发工程师了。一般线上出现问题

后会先抛给前端工程师，如果前端工程师定位到是后台的问题，再抛给后台工程师，后台定位后也可能抛给底层开发工程师，甚至追溯到运维工程师处。在这个过程中，有几个问题需要注意，开发过程常常是前、后台并行开始的，那么在开发中，前端如何在没有后台数据接口的情况下进行开发协作呢？

### 📌 数据协议文档

在需求开发之前，我们通常会通过文档来定义数据接口协议，这是很好的习惯，但实际上这种方式存在问题。工程师一旦开始开发，可能就没有太多时间去更新维护接口文档了，而新的数据接口在开发过程中默认是一定会变的，这样即使使用了接口文档管理系统也不一定实用，所以尽量推荐使用 RESTful 的通用协议规范来定义数据接口。当然基本的文档一定是需要的，它可以帮助我们解决大部分的问题。

### 📌 开发沟通方式

这点直接决定前、后端工程师开发协作的效率。开发流程在进行时，必要的会议不能少，信息沟通的方式当然最好是前、后台开发人员能在相同的办公区域内协作。一些大的企业由于组织架构原因，前、后端工程师可能会分开办公，这种情况下也要尽量采用最直接的沟通方式来节省沟通时间。

## 5.7.4 与运维工程师的“周旋”

前端工程师很少会和运维人员沟通，但也不是完全没有。运维人员往往很有特点，由于涉及稳定性问题，所以处理问题比较谨慎。如果你要找运维工程师协作，一定要有耐心，同时也不能被他们拖慢节奏，也许你唯一能做的就是 Push，推动他们配合你的工作。当然如果分工比较综合或团队比较小，也可能不存在这些问题。

## 5.7.5 对前端团队的支持

作为一名前端工程师，除了保质完成业务外，也要利用一部分时间来不断学习，支持团队的技术建设，例如团队开源项目开发维护等，自己如果想尝试新的想法也可以提出。在业务的开发过程中，我们常常充当前端工程师的角色，而在团队技术影响力的建设中，我们也肩负着团队技术驱动者和突破者的使命。这样前端团队才能越来越有影响力，相反则会在前端的学习发展中渐渐失去成就感，甚至失去工作的激情。所以，希望你能成为别人眼中的优秀工程师，同时在团队的技术建设中体现自身的价值。

## 5.8 本章小结

在这一章中，我们围绕前端项目的实践技术重点向读者们介绍了前端开发规范、组件规范设计、自动化构建原理、前端性能优化、前端数据分析和 SEO 等内容。深入学习和理解它们的设计思路和原理能帮助我们解决前端大型项目开发时遇到的问题，同时这些也是前端工程师进行自我提升的必备素养，是前端学习过程中最重要的技术实践内容，希望读者们可以有一个通透的理解。下一章中，我们将开始分析前端的跨栈技术，带领大家学习前端知识技术在后端和客户端中的应用和实践。

# 第 6 章

## 前端跨栈技术

随着互联网架构的不断演进，前端技术框架从后台输出页面到后台 MVC，再到前端 MVC、MVP、MVVM，以及到 Virtual DOM 和 MNV\*的实现，已经发生了巨大的变化。整体上来看，前端也正在朝着模块化、组件化和高性能 Web 开发模式化的方向快速发展。除了传统桌面浏览器端 Web 上的应用，前端技术栈在服务端或移动端上的尝试和发展也从来没有停止过，而且形成了一系列成熟的解决方案。所以无论如何，可以肯定的是，前端的技术栈能解决的绝不只是页面上的问题，前端工程师的追求也绝不只是页面上的技术。前端技术栈到底如何实现跨服务端，又如何扩展到移动端开发呢？这些将是我们本章要具体讨论的。

### 6.1 JavaScript跨后端实现技术

这几年全栈工程师已然成为一个很热门的关键词，从最早的 MEAN 技术栈到后端直出，再到现在的前后端同构，前端通过与 Node 结合的开发模式越来越被开发者认同并在越来越多的项目中得到实践。前端开发者都热衷于在 Node 上开发有以下几个原因。

- Node 是一个基于事件驱动和无阻塞的服务器，非常适合处理并发请求，因此构建在 Node 上的应用服务相比其他技术实现的服务性能表现要好，尽管目前 Node 服务器仍然是单进程运行的。
- Node 端运行的是 JavaScript，对于前端开发者来说学习成本较低，要关注的问题相对来说比前端更纯粹些，例如前端 JavaScript 兼容问题或者性能问题都不需要过于关注（不是说 Node 上就不用关注 JavaScript 性能问题，只是浏览器端 JavaScript 的性能问题显得更为突出）。

- 作为一名前端工程师确实需要掌握一门后台语言来辅助自己的技术学习。
- Node 端处理数据渲染的方式能够解决前端无法解决的问题，这在大型 Web 应用场景下的优势就体现出来了，这也是目前 Node 后端直出或同构的实现方式被开发者广泛使用的一个重要原因。

### 6.1.1 Node后端开发基础概述

作为前端工程师，我们要进行 Node 端的应用开发，必须先了解一些基础的知识。我们先来看一下进行 Node 后端上的应用开发通常需要具备哪些方面的基础知识和技术。

从图 6-1 中可以看出，要掌握 Node 后端的主要开发技术对前端工程师来说并不复杂，很多知识和技术都和前端内容类似，而且相对来说大多是比较纯粹的逻辑实现。

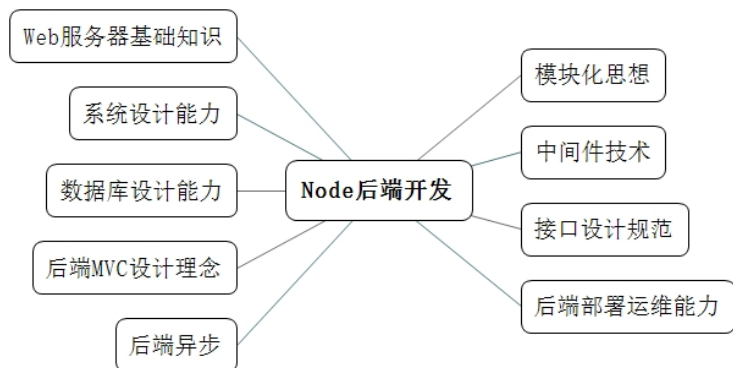


图 6-1 Node 后端基础知识和技术

- 服务器知识基础。和浏览器端开发一样，我们需要对 Web 服务器的一些基础知识有较全面的认识，例如 HTTP 请求的过程、返回码、缓存、Cookie 或 Session 在服务器端的工作机制、Web 安全（sql 注入、xss 内容、用户认证信息加密等处理方式）等问题，我们都应该有较清晰的了解。幸运的是，这些和前端的基础知识是相通的。
- 简单的数据库设计能力。少数情况下，我们可能会自己去设计一些简单数据库存储表的结构，这就要求我们具备简单设计数据库结构和数据库表的能力，目前在 Node 上结合 MongoDB 等 NoSQL 数据库，使用起来已经非常方便了。结合 MongoDB 对数据库对象的常用操作进行抽象处理的代码如下。

```
const connectName = 'localhost/datasite';  
const dbconnect = require('monk')(connectName);
```

```
function DB(dbname) {
  this.db = dbconnect.get(dbname);

  // 插入数据库记录操作
  this.insert = function*(data) {
    let result = yield this.db.insert(data);
    return result;
  }

  // 查找数据库记录操作
  this.find = function* (obj, query) {
    let result;
    if (query) {
      result = yield this.db.find(obj, query);
    } else {
      result = yield this.db.find(obj);
    }
    return result;
  }

  // 更新数据库记录操作
  this.update = function* (condition, data) {
    let result = yield this.db.update(condition, data);
    return result;
  }

  // 删除数据库记录操作
  this.remove = function* (condition) {
    let result = yield this.db.remove(condition);
    return result;
  }
}

module.exports = DB;
```

- 后端 MVC 设计理念。就 Node 端 Web 框架来说，目前主流设计模式仍是 MVC 方式，即用户请求进入路由层 Route 匹配，匹配成功后进入控制器 Controller，控制器调用 Model 数据库操作，然后将处理后的数据结合 View 模板渲染出 HTML 文本给用户，或者是将处理后的数据直接作为接口返回。但其实这个流程和前端 MVC 框架的实现理念也是类似的，只是实现的技术不同而已。
- 后端异步。除了理解异步的概念，我们对 Node 服务端异步编程的方式也要非常熟悉。例如对数据库操作或网络请求的异步处理方式我们要有清晰的理解。同时，Node 服务器对 ECMAScript 6+ 的支持为我们实现异步编程提供了许多便捷的实现方案，如

Promise、Generator、async/await 等，这些都是之前重点介绍过的，所以不用太担心技术实现的问题，例如新的 Koa 框架就支持 async/await 来直接处理 Web 后端中的异步逻辑。

```
const Koa = require('koa');
const app = new Koa();

app.use(async (ctx, next) => {
  const start = new Date();
  await next();
  const ms = new Date() - start;
  // 输出请求地址 URL 和所用时间
  console.log(`${ctx.url} : ${ms}ms`);
});

// 页面响应输出
app.use(ctx => {
  ctx.body = 'Hello Koa!';
});
```

- 模块化思想。目前的 Node 端 JavaScript 开发主要以 ECMAScript 6+ 标准为主，文件之间逻辑的引用仍是通过模块化机制来实现的，所以这里理解模块化规范和使用仍是非常重要的，在前面章节中我们也讲到，Node 端常用的模块化规范以 CommonJS 和 import/export 为主，与浏览器开发的常用模块化规范是相同的，这里不再赘述。
- 中间件技术。在实际开发过程中，我们还需要熟练掌握常用中间件的运用和开发。例如 Cookie、Session、Body 解析等高效的中间件能够帮助我们提高开发的效率，不过在特殊的场景中，我们也需要根据具体问题来开发满足具体需求的中间件模块，例如在 Koa 中就可以用如下方式引入自己的中间件模块。

```
const koa = require('koa');
const app = koa();
const myMiddleware = require('./middleware/my-middleware');

app.use(myMiddleware);
```

- 接口设计规范。推荐使用如 RESTful 这样的规范来定义数据接口，前面章节中我们也具体介绍过 RESTful 接口规范设计的好处，所以要尽量利用它的优势来解决问题。
- 后端部署技术和基本运维能力。因为 Node 主要是在服务器上工作的，所以掌握基本的服务器部署和运维能力也是很必要的，例如 Log 日志、服务器性能数据的收集和查看等，都有助于我们及时发现和定位服务器端的脚本运行问题。

需要了解的是，这里讲到的主要是针对服务器 Web 层的实现技术，关于底层服务层数据开发设计的内容本书不涉及。

### 6.1.2 早期MEAN简介

Node 出现的早期还不像现在一样拥有很复杂的概念，相关技术和语言的标准还不成熟，Node 开发一般用的比较多的方案就是使用 Express 作为 Web 框架进行小型的 Web 站点建设，与之结合的主流技术则以 M(Mysql)、E(Express)、A(Angular)、N(Node)最为典型，甚至到了今天 MEAN 技术组合的方式仍在沿用。图 6-2 为 MEAN 的经典结构。

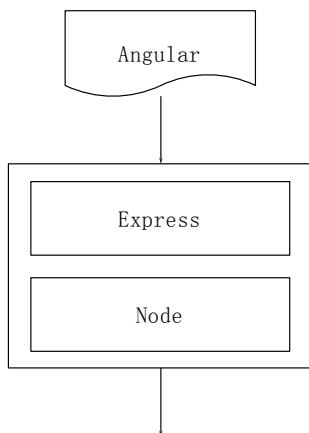


图 6-2 MEAN 全栈架构示意图

前端一般使用 Angular 来管理实现页面应用，服务端 Web 框架以 Express 为主，同时使用免费开源的 MySQL 数据库，这样就可以很快地构建一个 Web 应用了。关于 MEAN 的使用也非常简单，读者们参看对应文档就可以快速入门，MEAN 的安装使用方法如下。

```
$ npm install -g mean-cli
$ mean init appName
$ cd appName && npm install
$ gulp
$ node server // 打开浏览器访问 http://localhost:3000/就可以启动运行一个简单的 MEAN 应用了
```

今天我们可能不一定再去选择使用它，因为可以代替实现的成熟方案已经很多了，各类其他前后端框架都可以用来灵活组合作为 MEAN 的替代选型方案。



### 6.1.3 Node后端数据渲染

对于前端开发者来说,在大型 Web 应用开发中,很多时候并不需要完全重新设计整个应用后台的架构,更多的情况下需要结合 Node 的能力帮助我们解决前后端分离开发模式下无法解决的问题。我们先来看下通常前后端分离的开发模式下有哪些问题,利用 Node 端的服务又是如何帮助我们解决这些问题的。

#### 📌 SPA场景下SEO的问题

通常情况下,SPA 应用或前后端分离的开发模式下页面加载的基本流程是,浏览器端先加载一个空页面和 JavaScript 脚本,然后异步请求接口获取数据,渲染页面数据内容后展示给用户。那么问题来了,搜索引擎抓取页面解析该页面 HTML 中关键字、描述或其他内容时,JavaScript 尚未调用执行,搜索引擎获取到的仅仅是一个空页面,所以无法获取页面上<body>中的具体内容,这就比较影响搜索引擎收录页面的内容排行了。尽管我们会在空页面的<meta>里面添加 keyword 和 description 的内容,但这肯定是不够的,因为页面关键性的正文内容描述并没有被搜索引擎获取到。

如果使用 Node 后端数据渲染(有人称之为直出,后文中也称之为直出层),在页面请求时将内容渲染到页面上输出,那么搜索引擎获取到的 HTML 就已经包含页面完整的内容,页面也就更容易被检索到了。

#### 📌 前端页面渲染展示缓慢的问题

除了 SEO 问题,在前后端分离的开发模式下页面在 JavaScript 执行渲染之前是空白的(或提示用户加载中)。如图 6-3 所示,用户在看到数据时已经花费的网络等待时间:DOM 下载时间 + DOM 解析时间 + JavaScript 文件请求时间 + JavaScript 部分执行时间 + 接口请求时间 + DOM 渲染时间。这时用户看到页面数据时已经是三次串行网络资源请求之后的事情了。



图 6-3 前后端分离方式页面渲染主要流程

然而,如果使用后端直出来进行数据渲染,首先 SEO 的问题不复存在,用户浏览器加载完 DOM 的内容解析后即可立即展示,网络加载的问题也得到解决。其他的逻辑操作(如事件绑定和滚动加载的内容)则可按需、按异步加载,从而大幅度减少展示页面内容花费的时间。那么一般 Node 后端数据渲染的整个流程又是怎样的呢?

图 6-4 为目前一般后台页面数据直出的通用架构设计，直出层接受前端的路由请求，并在 Node 端的 Controller 层异步请求服务接入层接口，获得 Model 数据并进行组装拼接，然后提取相对应的 Node 端 View 模板渲染出 HTML 输出给用户浏览器，而不用通过前端 JavaScript 请求动态数据后渲染。不仅如此，直出层根据不同的浏览器 userAgent，也可以提取不同的模板渲染页面返回给不同的用户浏览器，所以这种实现方式不仅非常适合大型应用服务的实现场景，而且可以方便地实现网站的响应式内容直出。

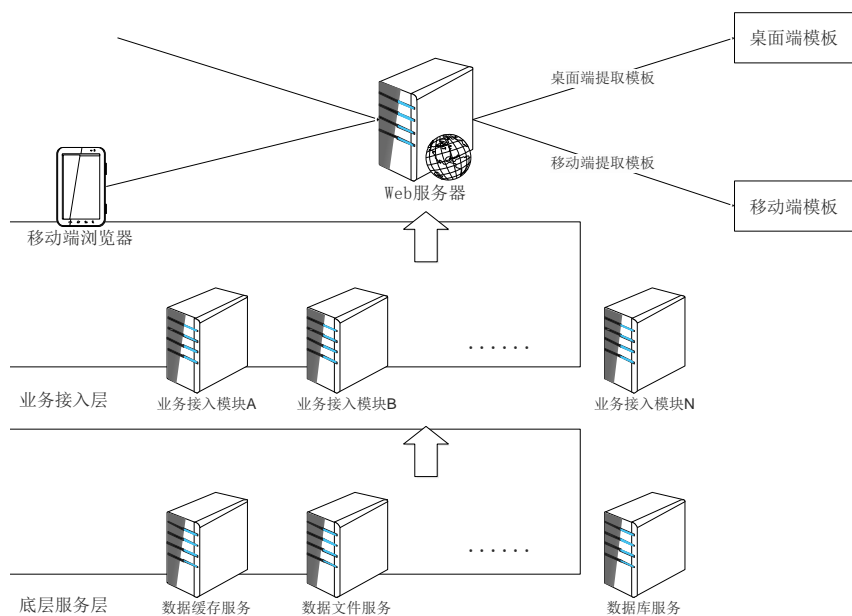


图 6-4 Node 直出层开发 Web 架构

### 6.1.4 前后端同构概述

在前后端分离的开发模式上加入直出层，解决了 SEO 和数据加载显示缓慢的问题。可是我们不得不思考两个新的问题。

- 前端的开发实现向直出层偏移，我们不得不在原来的开发模式上做出修改来适应直出层内容的开发，例如修改后端模板来适应现有的开发模式，结果我们不得不维护两套不同的前后台模板或技术实现——前端渲染实现逻辑和后端直出实现逻辑，尽管可能都是用 JavaScript 写的。
- 如果是在移动端 Hybrid 应用上，离线包机制实现可能就会出现问题。因为每次都是从

后端直出 HTML 结构给前端，这样就难做到将 HTML 文件进行离线缓存，而只能进行其他静态文件的缓存。在 Hybrid App 的应用场景下，其实我们更希望做到的是移动端首次打开页面时使用后端直出内容来解决加载慢和 SEO 问题，而在有离线缓存的情况下则使用客户端本地缓存的静态文件拉取数据返回渲染的方式来实现，或者未来在高版本的浏览器支持 HTTP2 的条件下使用前端渲染，低端浏览器不支持 HTTP2 的情况下则使用直出的方式实现。

所以我们需要一套完善的开发方式，和原有开发方式保持一致，且能够同时用于前后端分离的开发模式和后端数据渲染模板开发方式中。这种开发模式就是我们所说的前后端同构，下面系统地来了解一下前后端同构的一些内容。

### 1. 实现同构的核心

前后端同构的宗旨是，只开发一套项目代码，既可以用来实现前端的 JavaScript 加载渲染也可以用于后台的直出渲染。为什么可以这样做呢？和前端渲染数据内容的方式相同，页面直出层内容也是通过数据加上模板编译的方式生成的，前端渲染和后台直出的模式生成 DOM 结构的区别只在于数据和模板的渲染发生在什么时候。如果使用一套能在前端和后端都编译数据的模板系统，我们就可以做到使用同一套开发代码在前后端分别进行数据渲染解析。因此前后端同构的核心问题是实现前后台数据渲染的统一性。

### 2. 同构的优势

可以确定的是，除了解决前后端开发方式的问题，前后端同构的网站具有一些明显的优势：（1）可以根据用户的需求方便地选择使用前端渲染数据还是后台直出页面数据；（2）开发者只需维护一套前端代码，而且可以沿用前端原有的项目组件化管理、打包构建方式，根据不同的构建指令生成类似的前后端数据模板或组件在前后端执行解析，所以这对于 DOM 结构层上的开发方式应该是一致的。

当然，同构的目的是为了统一前后台的数据渲染方案，自然也会牵扯到前后端的适应性修改，实现前后端同构要做的一些额外工作可能就是前端工程师要开发 Node 端直出层上的路由配置和实现数据接口的编写。从实践的经验上来看，这部分的工作量常常是无法避免的，但是配合在直出层来进行会更加值得。

## 6.1.5 前后端同构实现原理

讲到前后端同构的实现，我们或许会很快想到一些提供这项能力的框架，但我们并不讨论

具体的框架，而是讨论实现的原理。目前就前后端同构的技术实现上来看，至少有三种思路：数据模板的前端渲染和后台直出、MVVM 的前端实现和后台直出、Virtual DOM 的前端渲染和后端直出。下面逐一介绍。

### 基于数据模板的前后端同构方案

早在前端 MVC 开发的时代，前端模板的使用就非常广泛，例如 Mustache、Handlebar 等，基本原理是将模板描述语法与数据进行拼接生成 HTML 代码字符串插入到页面特定的元素中来完成数据的渲染。同理，后端直出层也可以通过该方法来实现数据的渲染产生 HTML 字符串输出到页面上，如果前后端使用同一个模板解析引擎，那么我们只需要编写同一段模板描述语法结构就可以在前端和后端分开进行渲染了。

如图 6-5 所示，对于前端开发的同一段模板语法结构，我们既可以选择在浏览器端渲染生成 HTML 字符串输出，也可以选择在后端渲染生成 HTML 字符串输出。如果选择在前端渲染，则可以将模板进行打包编译，在数据请求成功后进行 DOM 渲染；如果选择后端渲染，就可以将模板数据直接发送到直出层的 View 视图进行渲染，实现同一个模板语法结构在前后端渲染出相同的内容。不过这里需要注意的事是，要保证前后端使用的模板渲染引擎或者模板解析的语法是一致的。

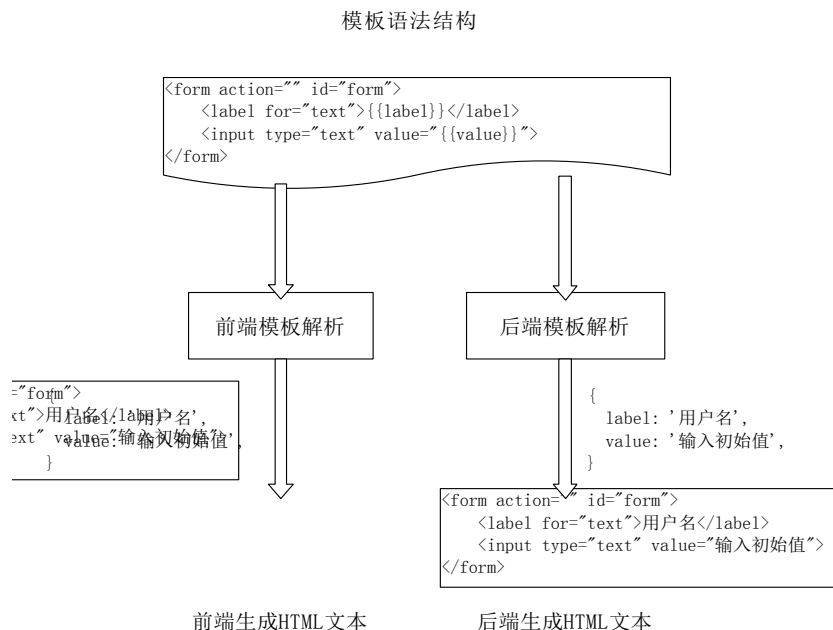


图 6-5 模板渲染的同构方式原理

## 基于MVVM的前后端同构

我们知道 MVVM 框架页面上的 JavaScript 逻辑主要是通过 Directive（不只是 Directive，还有 filter、表达式等，以 Directive 为主）来实现的，一般前端页面加载完成后会开始扫描 DOM 结构中的 Directive 指令并进行 DOM 操作渲染或事件绑定，所以数据的显示仍然需要页面执行 Directive 后才能完成。那么如果将 Directive 的操作在直出层实现，浏览器直接输出的页面不就是渲染后的内容数据了吗？

如图 6-6 所示，在项目开发中，前端编写的同一段 MVVM 的语法结构通过前端 MVVM 框架解析或后端 Directive 运行解析最终都可以生成相同的 HTML 结构，不同的是前端执行解析后生成的是 ViewModel 对象并通过浏览器体现，后端渲染则生成 HTML 标签的文本字符串输出给浏览器。这里同样需要做一件事，即在后台实现一个与前端解析 Directive 相同的模块，甚至还包括一些 filter、语法表达式等的实现。这样就可以在前后端完成同一段语法结构的解析了。

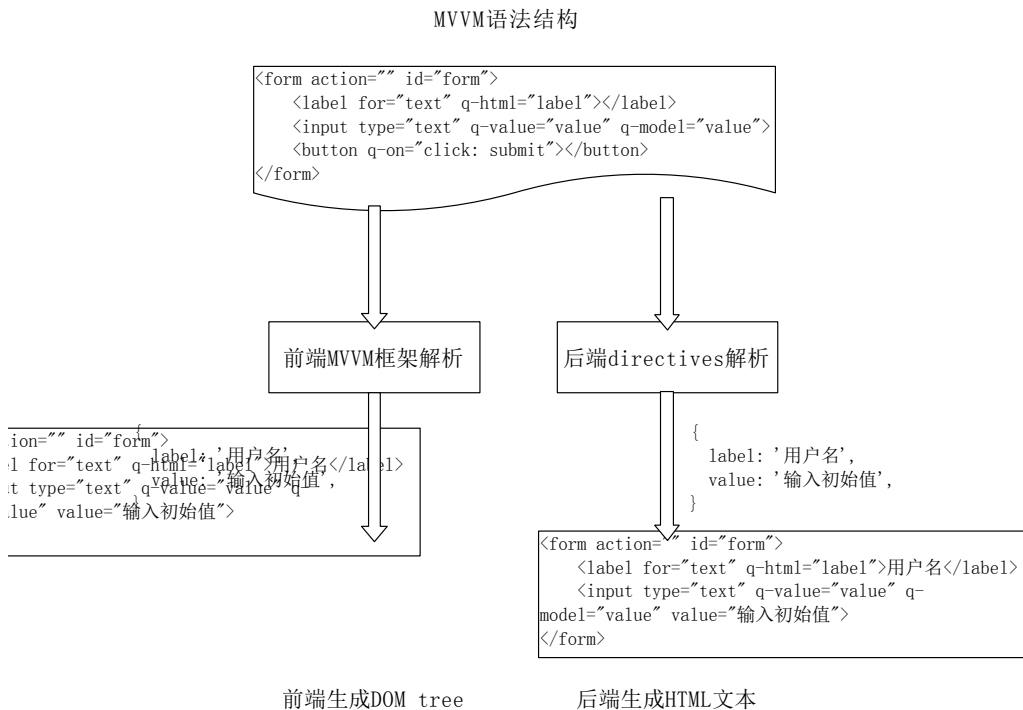


图 6-6 MVVM 实现的同构方式原理

## 基于Virtual DOM的前后端同构

前面讲到，目前 Virtual DOM 作为一种新的编程概念被广泛应用在实际项目开发中，其核

心是使用 JavaScript 对象来描述 DOM 结构。那么既然 Virtual DOM 是一个 JavaScript 对象，就表示其可以同时存在于前后端，通过不同的处理方式来实现同构。

如图 6-7 所示，在前端开发的组件中声明某段 Virtual DOM 描述语法，然后通过 Virtual DOM 框架解析生成 Virtual DOM，这里的 Virtual DOM 既可以用于在浏览器端生成前端的 DOM 结构，也可以在直出层直接转换成 HTML 标记的文本字符串输出，后面这种情况就可以在服务端上实现 Virtual DOM 到 HTML 文本字符串的转换。这样，通过对 Virtual DOM 的不同操作处理，就可以统一前后端渲染机制，实现组件的前后端对同一段描述语法进行渲染。

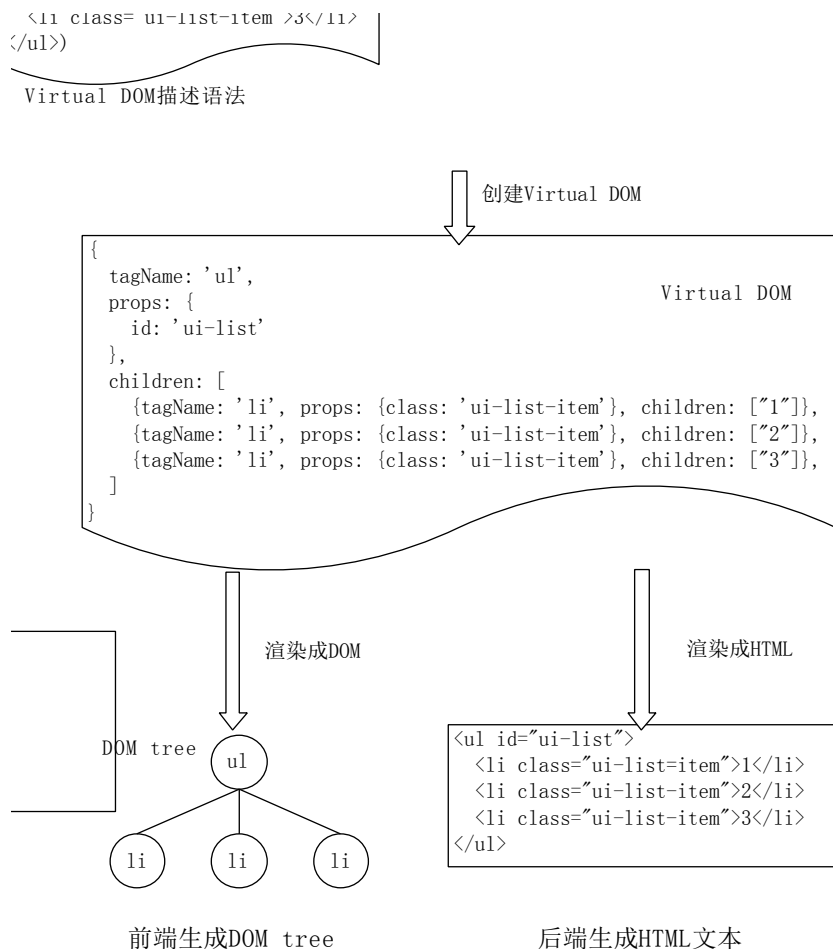


图 6-7 Virtual DOM 实现同构方式原理

这里 Virtual DOM 上的逻辑实现仍然需要在浏览器端进行事件绑定来完成，最好能让同构

框架帮助我们自动完成,根据 HTML 的结构进行特定的事件绑定处理,保证最后展示给用户的页面是完整且带有交互逻辑的。

这里介绍了三种前后端同构实现的思路,而且三种方法目前都有框架来实现。但是无论选择使用哪一种方案,其核心都是一致的——使用同一种结构内容的描述方式通过特定的规则解析转化成前端和服务端均能够处理的 DOM 结构形式。此时,无论是前端模板、ViewModel、Virtual DOM、DOM 还是 HTML 片段,都只是前端浏览器结构层内容的表示方式,而且是可以相互转化的,所不同的是,页面上的 DOM 是用户最终看到的内容。

图 6-8 为前后端同构的原理图。可以认为实现前后端同构的核心都体现在 HTML 的结构形式变化上,页面内容的描述方式有很多,而且可以通过特定的处理过程实现转化,这样就提供了更多的可能性。

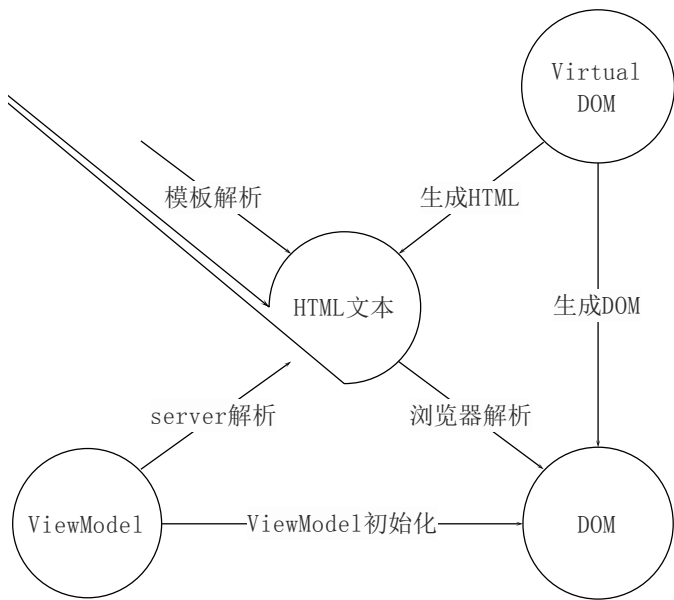


图 6-8 前后端同构的核心原理

需要注意的是,无论选择哪一种前后端同构的实现方案,所需要处理的几个问题是类似的。

- 前后端框架选择。这里主要包括前端使用的主要框架和后端结构渲染解析模块的选择,通常选择不同的实现方式所对应的框架实现也不一样。例如基于 MVVM 的实现和基于 Virtual DOM 的具体框架实现区别还是比较大的。

- 模板渲染机制。这对实现前后端内容渲染的统一性来说比较重要。刚刚也重点讨论过了，关键的不同点主要在 HTML 描述和转化方式的选择上面，我们需要保证前后端都能对同一套模板或描述语法进行识别处理，生成前后端各自能处理的结构。
- 构建打包。同一套代码基于前后端场景打包完成后是不一样的，并且对于开发者来说需要有完整的模块化机制、打包体系和不同的输出调试目录，方便业务层上的开发。打包完成后，服务端实现逻辑和前端实现逻辑也应该分开发布部署，也可以通过在直出层 Web 服务器上判断来选择使用哪种方式输出。也可能是将前端实现的逻辑作为移动端应用的离线包发布，服务端的实现逻辑完成直出层的 View 模板渲染，这样更易于管理。
- 渲染和直出区分。用户必须能够很方便地选择是使用前端渲染的方式还是后台直出的方式，例如可以在浏览器地址 URL 后面添加 `render=1` 参数来区分，如果带有 `render` 参数则使用前端渲染，同时进行事件绑定处理，否则默认统一使用后端直出的方式，前端不渲染数据，只做事件绑定处理。这样就可以很灵活地满足更多的应用场景。以基于数据模板的前后端同构的实现方式举例，页面中某个模块的渲染方式实现代码如下。

```
// 获取 URL 中的参数
let getUrlParam = function(name) {
  let reg = new RegExp("(^|&)" + name + "=(^&|*)(&|$)");
  let r = window.location.search.substr(1).match(reg);
  if (r != null) {
    return unescape(r[2]);
  }
  return null;
};

const component = new Component({
  // 前端某个具体模块渲染逻辑
  init (data) {
    // 如果 URL 中带有 render 参数，则使用前端模板渲染的方式展示数据，否则只做事件绑定
    if (getUrlParam('render')) {
      this._renderData(data);
    }
    this._bindEvent();
  },

  _renderData (data) {
    // 调用模板渲染数据
    this.$el.html(renderTpl({data: data}));
  },
});
```



```
    _bindEvent(){  
        // TODOS, 页面事件绑定  
    }  
});
```

以上为前后端同构实现的主要内容，相信大家对同构的三种主要实现方式的原理也比较清楚了。实现同构的形式很灵活，通常不只是我们知道的某一类框架实现的方式，所以我们可以结合更多实际场景来灵活选择。

从前端开发者的角度来看，使用 Node 进行服务器端开发的场景有很多，但主要还是以使用 Node 作为服务直出层最为典型。这一节主要就后端直出、前后端同构的原理进行了较全面的分析，实际项目中我们可以选择借鉴已有的成熟解决方案，也可以根据自己团队的特点来重新设计和实现项目同构开发整个流程中的具体细节。

## 6.2 跨终端设计与实现

### 6.2.1 Hybrid技术趋势

移动互联网兴起后，智能移动设备出现，大量应用市场的 Native 应用也开始涌现。随着第一波移动端互联网开发浪潮渐渐平静，各类 Native 应用开始进入有序更新迭代的阶段。人们对移动互联网需求急剧增长，Native 应用快速迭代开发的需求也越来越多，但是现有 Native 应用的开发迭代速度依然无法满足市场快速变化的需要。随之而来的是 HTML5 的出现，它允许开发者在移动设备上快速开发网页端应用，并让移动互联网应用开发很快进入了 Native 应用、Web 应用、Hybrid 应用并存的时代。关于 Native 应用、Web 应用、Hybrid 应用，相信大家都有所了解，但就目前的行业发展现状来看，它们之间的区别显然也有了一些变化，下面我们一起来看一看这三者更全面的对比。

#### Native 应用的优点

- 原生系统级 Native API 的支持，如访问本地资源、相机 API 等
- 资源在打包安装时生成，节省用户使用时的流量
- 可针对不同平台特性进行用户体验优化
- 运行速度快、性能好，可使用原生 Native 动画库

### Native 应用的缺点

- 开发成本高，兼容性差，尤其是对于 Android 机型
- 维护成本高，用户必须手动下载更新，历史版本也需要维护
- 上线时间不确定，一般需要通过应用商店的审核
- 版本更新慢，更新时需要用户重新下载安装包
- 应用界面的内容不可被搜索引擎检索

### Web 应用的优点

- 开发成本低，使用前端开发技术即可
- 跨平台和终端，基于浏览器或 WebView 运行
- 部署方式简单、快捷，无需用户安装
- 用户总能访问到最新版本，迭代速度快
- 内容可被搜索引擎检索

### Web 应用的缺点

- 浏览体验无法超越 Native 应用
- 消息推送、动画等实现方式相对 Native 实现方式较差
- 不能调用设备的原生特性，如无法访问本地资源、相机 API 等

### Hybrid 应用的优点

- 开发成本较低，可以使用前端开发技术，甚至可以自动添加 Native 外壳来实现独立移动端应用
- 跨平台和终端，内容网页可基于浏览器或 WebView 运行
- 拥有与 Web 应用相同的快速迭代特性
- 部署方式简单、快捷，只更新 Web 资源即可

- 可支持实现离线应用
- 可通过 JSBridge 调用设备的系统级 API，如访问本地资源、相机 API 等
- 原生应用版本迭代和 Web 功能迭代相互独立也可以相互结合
- 不同性能需求的功能可以选择性使用 Native 或 Web 实现
- 内容可被搜索引擎检索
- 借助于 MNV\* 的开发模式可以更接近 Native 应用的用户体验

### Hybrid 应用的缺点

- 部分机型兼容相对 Native 较差，但比 Web 应用体验好很多

显然，目前 Hybrid 应用的开发模式也已经突破了开发效率和性能的两大问题，更加适应移动互联网时代产品高迭代速度的需求，而且目前主流的移动端应用均是采用 Hybrid 的方式来实现的，所以为什么使用 Hybrid 就不言而喻了。下面我们直接来看一下 Hybrid 应用开发的几种常见实现技术。

## 6.2.2 Hybrid 实现方式

相比 Native 原生应用开发，Hybrid 应用开发的方式就比较多了，我们先来看看以前端开发实现为主的 Hybrid 开发方式。

### 👉 以前端为主的 Hybrid 实现方式

这种方法是以完全的前端模式来开发整个应用的，页面开发完成后，通过工具自动打包将前端资源目录装入 Native 容器中运行，如图 6-9 所示，打开应用运行时，除了部分通用的简单逻辑外，内部逻辑全部由打包的 Web 端代码来实现。在以 Apache Cordova 为基础的开发工具实现下，前端开发者不需要了解 Native 相关的内容，只需要专注于前端页面功能的开发即可，功能开发完成后自动将前端内容统一打包进发布安装包，安装时解压到移动端本地目录加载运行。

这种思路比较简单，通常借助框架即可实现，而且成本很小。其优势是前端开发者可以独立快速构建 Hybrid 应用，不需要 Native 开发人员的支持，前端调用 Native 的解决方案也可以使用开源的 JSBridge 库（如 cordova.js 等）来实现。但是这种方式缺点也很明显：Native 功能的实现只能通过 Web 的方式，并且无法添加复杂的 Native 功能；如果遇到即时通讯或服务端推送的应用场景，使用 Web 的方式实现性能就比较差；与 Native 的交互方式上也会受到固有开

源库实现的限制，无法灵活扩展；无法避免在应用版本更新时需要重新下载安装应用的问题；WebView 性能局限，所有的功能逻辑都是在 WebView 中实现的，在页面复杂情况下性能比较慢，移动端应用的 WebView 执行性能只有移动端浏览器性能的 1/3~1/4，而移动端浏览器本身的解析就相对较慢。所以这种实现方式适合于中小型需要快速完成开发的应用场景，如果是在用户量较多、实时性要求较高、应用需要快速持续迭代或者需要与扩展 Native 功能结合的应用场景中，就不适用了。

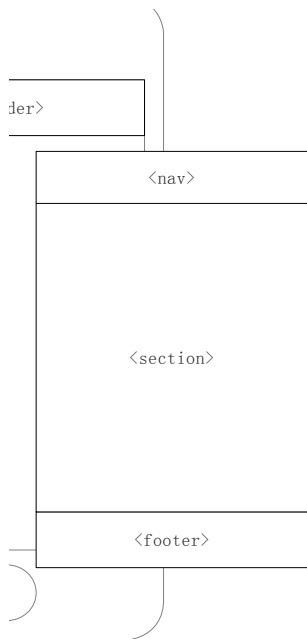


图 6-9 以 Web 内容为主的 Hybrid 实现方式

所以如果是应用复杂、功能较多、需要快速迭代更新的情况下，还是建议使用自定制 Native 和 Web 结合的 Hybrid 开发方式来组建应用。

### 👉 Native和Web结合的Hybrid方式

这里说的 Native 和 Web 结合的意思主要是指移动端应用中 Native 和 Web 功能上的结合开发实现，为什么需要这样做呢？通常在一个完整的移动端 Hybrid 应用中，功能是由核心 Native 功能和 Web 站点页面组成的，其中 Web 站点页面中可以调用 Native 功能。此外，Native 和 Web 的功能通常也不一样，实现自己最擅长的功能。例如 Native 就可以用来实现移动端应用的通用导航菜单、系统 UI 层、核心界面动效、默认访问界面、高效的消息推送或 APP 大版本的应用

更新等，因为这些功能一般比较稳定，变化不大，而且不涉及需要快速迭代的业务逻辑。而 Web 端则可用来实现开发迭代速度更快的相关业务层界面逻辑，它很可能是某个 Native 应用内关联的某个 Web 轻应用。

图 6-10 为某个支付应用 APP 的设计流程，进行网上商城购买商品时的功能业务最好用 Web 方式实现，如 APP 应用导航或支付流程等部分的界面就应该用 Native 来做。通过这种方式，可以保证关键性功能页面流程的用户体验和 Web 轻应用的快速迭代开发，更符合现代移动端应用的实现标准，目前国内一线互联网企业的移动端核心业务实现方案也都是使用这种结合方式来实现的。

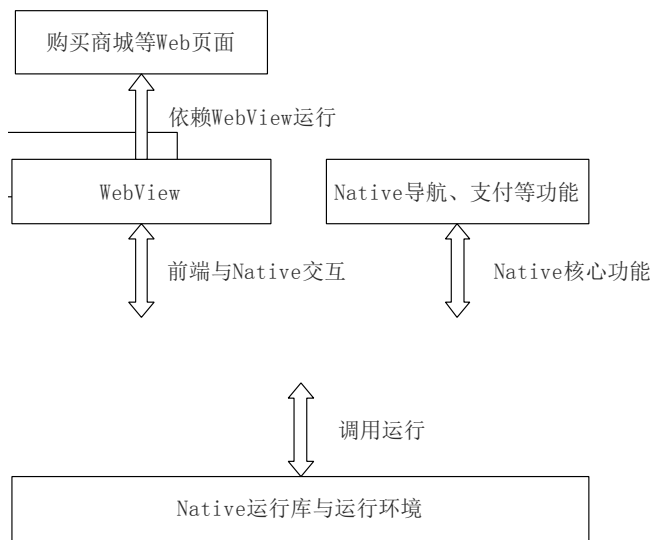


图 6-10 典型支付 APP 设计实现

需要注意的是，使用这种开发模式首先要非常关注开发过程中 Web 和 Native 的调用接口规范问题，因为这种情况下我们一般不会借助开源的交互实现方案，所以自己如何去设计 Web 和 Native 的交互协议就显得很重要了。本书第二章第五节向大家详细介绍了使用 JavaScript 与 Native 交互的协议与规范，目前一般的解决方式就是通过 JSBridge 协议在 Web 页面中调用 Native 功能。有关协议的设计规范和实现可以去参考此章节内容的介绍，这里就不再继续展开了。

### 6.2.3 基于localStorage的资源离线和更新技术

介绍完 Hybrid 应用的实现方式，我们再来重点看看 Hybrid 应用上前端资源离线和更新的问题。在 Hybrid 应用开发时，常常需要在离线的环境下打开页面或者为了让 Hybrid 页面应用

加载启动更快，避免长时间等待资源加载过程中造成页面空白的出现。因此我们必须要考虑使用资源的离线缓存技术来加快页面启动时的载入速度了。而且现代浏览器也提供了一些页面上静态资源文件级缓存与更新的方式，下面我们就来看一下 Hybrid 应用中实现 Web 端资源离线与更新的可行方案都有哪些。

### 📁 ServiceWorker的资源离线与更新

在第一章第二节中我们了解了可以通过浏览器 Application Cache 的方式实现页面资源的离线加载和更新。但这里需要注意的是，Application Cache 这种方案目前开始被浏览器标准弃用了，取而代之的是 ServiceWorker 这种离线技术实现机制。前面章节中我们对 ServiceWorker 也进行了具体分析，这里仍要注意目前 ServiceWorker 的浏览器兼容性支持很差，导致这种方案目前还不成熟，或者说是短期内仍不是一个可行的实践方案。

### 📁 localStorage资源离线缓存与更新

当然除了这种未来可能使用的 ServiceWorker 解决方案，目前实现前端离线缓存一种比较简单高效的方法就是使用 localStorage。早期的离线资源缓存通常也是使用这种方式来实现的，而且国内几个大型互联网企业的前端团队在移动端资源离线化方面都曾经尝试过这种方法，其基本思路是将 JavaScript、CSS 资源文件甚至是接口返回的数据资源缓存到浏览器的 localStorage 中，下次打开页面时不进行 JavaScript 和 CSS 资源的请求，而是直接通过 localStorage 读取内容，然后插入到页面中解析执行。这里需要注意的是，为了判断是加载线上静态资源还是从 localStorage 中读取资源，页面中 JavaScript 和 CSS 资源的加载方式通常都是动态创建标签加载或通过 eval 执行的，而且通常只有页面在第二次打开或之后加载静态资源的情况才可能从 localStorage 中读取。下面是使用 localStorage 缓存读取 JavaScript 资源的一个简单实现。

```
<!-- 服务器最新的版本可以在最新的 html 文件中写入 -->
<div id="versionStore" data-version="1.4"></div>

<script>
let scriptPath = 'server/path/',
    script = document.createElement('script'),
    newVersion = document.getElementById('versionStore').getAttribute('data-verion'),
    oldVersion = localStorage.getItem('version')

/* 如果有版本更新或者本地没有缓存，则请求新的 JavaScript 内容插入到页面中执行，同时保存到本地 localStorage */
if(newVersion > (oldVersion || 0)){
    $.ajax({
        url: `${scriptPath}main.${newVersion}.js`,
        type: 'get',
```

```
    dataType: 'text',
    success: function(content){
        script.innerHTML = content;
        document.appendChild(script);
        // 更新 localStorage 静态资源内容
        _updateLocalStorage(scriptPath, content);
    }
});
}else{
    /* 如果有缓存且未更新, 则直接读取缓存内容 */
    script.innerHTML = localStorage.getItem(scriptPath);
    document.appendChild(script);
}
</script>
```

这是个简单的例子, 在加载文件时, 页面新的版本号已经写到 HTML 页面上或者通过单独的接口请求获取, 页面脚本通过获取页面上的最新版本号和本地 localStorage 保存的旧版本号进行对比, 如果本地没有版本号或版本号较旧, 则加载最新版本的静态资源文件到页面上, 同时更新本地原有的 localStorage 缓存内容和版本号, 否则直接读取 localStorage 的静态资源内容到页面中解析执行, 基本的实现流程如图 6-11 所示。

这种实现方式的好处是比较简单, 不需要服务端和移动客户端平台的支持, 可以实现纯移动端应用的离线访问。当然缺点也很明显: 首先 localStorage 的大小有限制 (同域一般认为是 5M 以内), 同域名的 localStorage 存储的离线资源较多时很可能会内容超出, 容易出错, 需要通过资源替换策略来处理, 这样就比较麻烦; 其次如果用户手动清空 localStorage 会使离线资源失效, 这个问题基本上不能解决; 还有就是读取 localStorage 的速度其实是比较慢的, 尤其在移动端, 如果 localStorage 的内容较多, 返回的速度可能会比较慢。

当然, 使用这种方式是可以解决一部分问题的。例如 Hybrid 应用的页面通过分享到社交平台打开的情况下, 如果用户是第二次打开这个页面, 利用这种缓存方式就可以加快页面的载入速度。因此在快速实现离线缓存的方式上, 这是一种很简易实用的方法。我们再来看看这种情况下, 应该如何实现 localStorage 静态资源的更新。

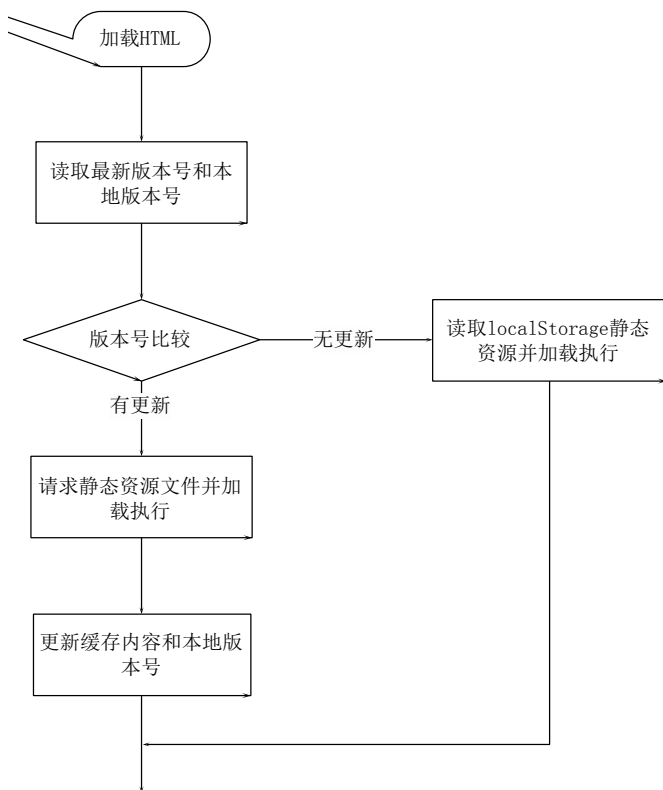


图 6-11 移动端常见页面结构

### 基于增量文件的更新方式

在上述资源访问和更新的过程中，当静态资源改变时，如果像上面那样每次都进行新文件的全量更新也不是不可以，很直接、易实现。但问题是，即使我们修改了代码中的一个字符或语句，就要更新整个静态资源文件。所以为了节省流量可以选择增量文件更新的方式。

要实现增量文件的更新需要做更多的事情。如图 6-12 所示，假如之前已经有 1.1、1.2、1.3 三个版本的脚本资源发布，而且目前使用各个版本的用户都存在，现在 1.4 版本的离线资源文件上传发布后，为了满足不同版本用户的增量更新，需要根据前面三个版本的文件内容与最新版本内容进行对比分析，分别生成三个不同版本的增量文件 1.1-1.4.js、1.2-1.4.js、1.3-1.4.js，同时还需要保留 1.4 版本的全量文件。此时服务器上 1.4 版本发布后就保留了四个不同的文件：1.4 版本文件、相对于 1.1 版本的增量文件、相对于 1.2 版本的增量文件、相对于 1.3 版本的增量文



件。新用户进入页面应用后直接拉取 1.4 版本文件即可，前三个版本的用户拉取的则分别是三个不同的增量文件。这样不同版本的用户就可以增量更新到不同的内容了，上一个例子的实现代码就可以如下编写了。

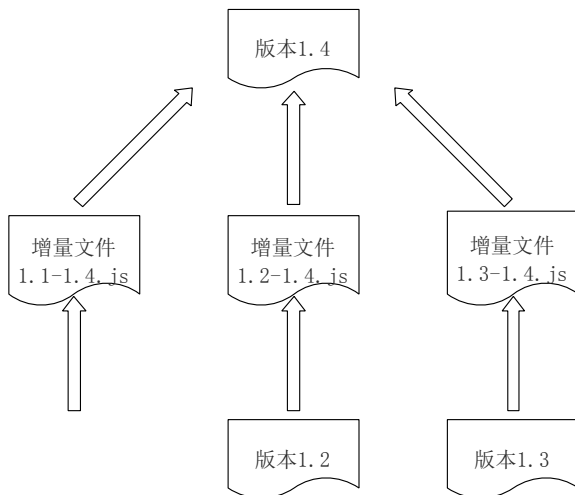


图 6-12 增量文件版本发布控制

```

<!-- 服务器最新的版本可以在最新的 html 文件中写入 -->
<div id="versionStore" data-version="1.4"></div>

<script>
let scriptPath = 'server/path/main.js';
let script = document.createElement('script');
let newVersion = document.getElementById('versionStore').getAttribute('data-verion');
let oldVersion = localStorage.getItem('version'); // 获取旧的版本号

if(oldVersion && newVersion > oldVersion){
  /*如果本地有缓存，且有版本更新，则请求新的对应版本的增量文件，然后进行增量计算，并生成的最新版本内容
  保存到本地 localStorage*/
  $.ajax({
    url: `${scriptPath}${oldVersion}-${newVersion}.js` ,
    type: 'get',
    dataType: 'text',
    success: function(content){
      // 根据增量文件内容和当前内容计算新的静态资源
      content = _calculate(content, localStorage.getItem(scriptPath));
      script.innerHTML = content;
      document.appendChild(script);
      // 更新 localStorage 静态资源内容
      _updateLocalStorage(scriptPath, content);
    }
  })
}

```

```
});  
}else if(!oldVersion){  
    // 本地没有缓存则直接获取最新全量文件  
    $.ajax({  
        url: `${scriptPath}main.${newVersion}.js` ,  
        type: 'get',  
        dataType: 'text',  
        success: function(content){  
            script.innerHTML = content;  
            document.appendChild(script);  
            // 更新 localStorage 静态资源内容  
            _updateLocalStorage(scriptPath, content);  
        }  
    });  
}else{  
    // 如果有缓存且没有版本更新, 则直接读取缓存内容  
    script.innerHTML = localStorage.getItem(scriptPath);  
    document.appendChild(script);  
}  
</script>
```

通过比较不同版本就可以只加载不同版本的增量文件,但同时需要在服务器端每次新版本发布时维护多个增量文件来适应不同的旧版本更新的需要。至于如何根据两个新旧版本的文件生成字符级的增量文件,这就是我们下面要继续讨论的内容了,目前主要有两种基本的算法来实现这一过程。

### 📁 基于文件代码分块的增量更新机制

这种思路是基于代码文件分块更新的增量算法,如图 6-13 所示,main.1.3.js 的文件字符串由几个字符串块连接组成,chunk1-chunk2-chunk3-chunk4(每个 chunk 代表分割后不同的代码字符串),此时需要在 chunk1 和 chunk2 之间添加 data1,chunk3 的内容被修改成了 chunk5,chunk4 的块被删除。新的代码文件字符串应该为 chunk1-data1-chunk2-chunk5。为了解决这个问题,我们用一种描述规则来表示每个代码文件块的变化,比如使用原有的序号 1 表示原来 chunk1 的内容不变化,加入 data1 块内容表示插入的新内容,-4 表示删除 chunk4 的文件块,那么就可以用数组[1, data1, 2, chunk5, -4]来表示新的增量文件了,具体表示: chunk1 未修改,后面拼接 data1, chunk2 未修改, chunk3 被移除,后面拼接 chunk5, chunk4 被移除掉。当浏览器下载到带有这个版本的增量文件时,就会在前一版本的代码文件字符串上根据这个规则修改,得到更新后的静态资源文件字符串内容并重新写入到 localStorage 中。

我们再来看一个具体的例子,新旧两个版本的 JavaScript 文件压缩后上线的代码字符串分别为 let a=1;alert(a);和 let a=1;alert(a+1);,我们根据块大小为 4 个字符来分割,

可以得到如图 6-14 所示的表示，所以计算获取的增量内容描述为[1,2,3,'t(a+', '1) ; ' ]。这种方式在代码较多的情况下就可以用块序号来描述不变的代码块，减小增量文件大小，达到节省流量的目的。

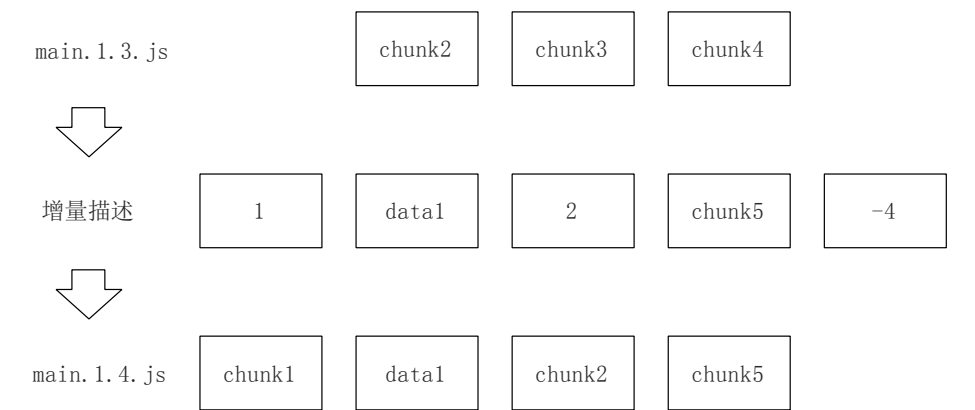


图 6-13 基于分块的增量文件算法思路

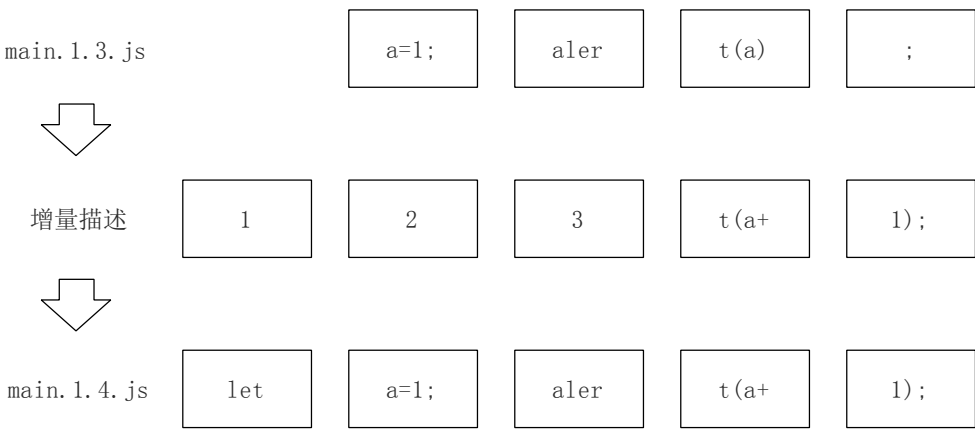


图 6-14 基于分块的增量文件算法案例

🔗 基于编辑距离的增量更新机制

前一种方法是基于文件内容分块 chunk 算法来进行增量更新的，节省资源的量取决于块的大小和内容变化的块序号分布，比如每个块中都只更新了一个字符，那么仍然需要下载这个文件的所有块内容。而根据编辑距离算法增量更新的方式就可以真正做到字符级别的更新了。

1965 年俄罗斯科学家 Vladimir Levenshtein 提出了 Levenshtein Distance（编辑距离，相关算法的实现内容比较多，有兴趣可以参考其他资料）的概念，它指的是从一个字符串变换到另

一个字符串所需要的最少变化操作步骤。那如果能计算获取两个文件对比变化时每个字符的操作步骤，就可以将操作步骤作为增量文件下载，然后在浏览器端进行代码的运算更新了。不过这种情况对于少量的字符更新很有用，如果一次更新的内容很多，生成的增量文件很有可能比源文件还大，所以实际使用过程中需要结合具体情况在这两种增量算法实现中进行选择。

## 6.2.4 基于Native与Web的资源离线和更新技术

### 📁 Native和Web结合的Hybrid资源离线与更新

接下来介绍另一种 Hybrid 应用上的离线实现机制，与 `localStorage` 上实现文件资源离线的方式不同的是，在这种 Native 和 Web 结合的 Hybrid 开发模式下，Web 端的代码资源是通过离线包的方式发布到服务端静态目录上的，同时主站点会保存一个线上的前端页面实现供浏览器直接加载使用。

如图 6-15 所示，通常 Native 应用启动时会主动拉取线上 Web 离线包版本，然后与本地保存的版本进行对比，如果没有更新则不做操作；如果本地的离线包需要更新或者本地没有离线包，则会去离线包服务器拉取最新的离线包或者拉取增量离线包到本地，然后解压合并到本地的指定离线包目录下。当有用户访问目标页面时，Native 应用会先检查该文件地址映射到的离线包本地目录中的文件，如果离线包目录有内容，则直接读取离线包目录的文件加载；否则线上拉取静态资源到页面上解析执行，同时通知 Native 应用去拉取最新的离线包资源，这样当下一次继续请求目标页面时 `WebView` 就可以读取到本地离线目录中的内容了。这也就是为什么通常我们拉取到的新离线包需要在下一次访问才生效的原因。另外，对于离线包的更新检查时机，也有的应用设计是在打开目标页面时，无论是否加载到离线内容，都会去检测是否下载新的离线包资源。

前端离线包的生成比较简单，一般是通过构建工具将站点资源目录直接压缩，在发布前端页面与静态资源的同时上传离线包到服务器或 CDN 上。这里需要注意的是，为了保证移动端应用每次尽可能拉到较小的离线包内容，上传服务器端离线包时也需要生成与之前版本对比的增量更新包发布到服务器或 CDN 上的。

增量包的计算方法与 `localStorage` 的增量文件计算方法类似，如图 6-16 所示，假如有 1.1、1.2、1.3 三个版本的离线包，当 1.4 版本的离线包上传后，需要对比前面三个版本的离线包内容分别生成三个不同版本的新增量包，同时保留 1.4 版本的全量离线包。此时服务器上 1.4 版本发布后就存在四个离线包：1.4 版本全量包、相对于 1.1 版本的增量包、相对于 1.2 版本的增量包、相对于 1.3 版本的增量包。这样新用户进入应用会直接拉取 1.4 版本全量离线包、前三个版本的

用户拉取的则分别是三个不同的增量包。这样不同版本的用户更新就没有问题了。至于如何根据两个离线包计算增量包的算法也和计算增量文件的算法类似，既可以根据增量包内每个文件资源的增量文件来计算，计算方法和 `localStorage` 实现增量文件的方法类似，也可以通过直接针对压缩包文件二进制数据内容分块进行对比的增量方法来实现。

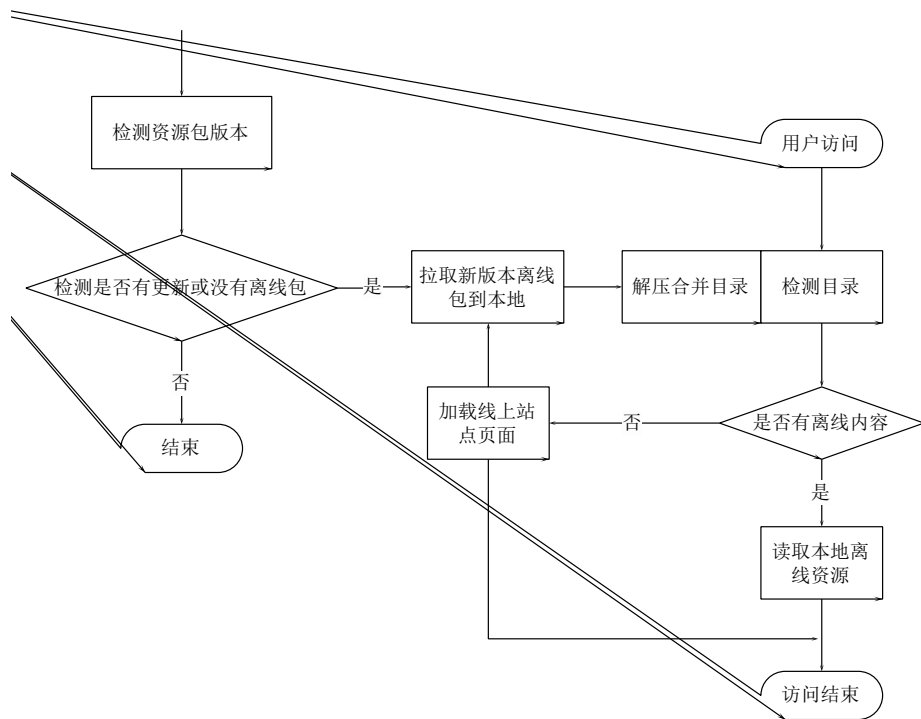


图 6-15 增量包更新机制

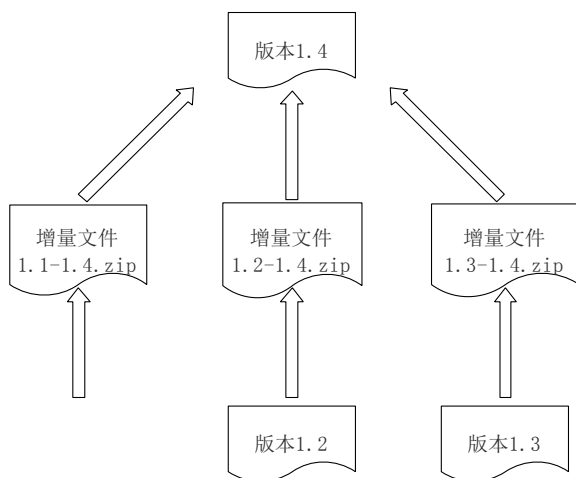


图 6-16 增量包产生算法

到此，离线包的运行思路已经分析完了。很多情况下，尤其是在大的互联网公司中，这些内容基本是后端或之前的同学已经设计好的，但是我们也要理解，如果是自己搭建实现这个流程的话，各个环节应该怎样去做，这是很重要的。

## 6.2.5 资源覆盖率统计

上面讲到，既然有了前端资源的离线和更新机制，那就要考虑在每次新资源包发布后统计新版本的更新覆盖率。这和没有离线的时候是有区别的，前端发布的新版本常常会在用户拉取到线上新版本资源后的第二次访问时才生效，那么在新的资源发到服务器上后，用户客户端里面可能仍会存在旧版本的运行代码逻辑，这就需要知道有多少用户已经更新到了新的版本。在增量包的生成过程中，如果某个旧版本的用户使用率已经很小，或者基本接近 0，那我们就可以考虑后面不再对这个版本生成增量包，并让这部分用户直接拉取最新的全量包，避免在版本发布较多时线上有过多的历史增量包版本存在。因此，统计离线包资源覆盖率是很有意义的。

统计的方法很多，一个简单可行的方式就是后台统计上报版本号。为了更加直接地体现版本分布的情况，我们可以忽略用户量上的统计，直接对访问量进行计算。在发布了新版本后，每次 PV 统计时带上版本号，最后根据 PV 中的版本号来统计访问不同版本上用户的分布情况。

图 6-17 显示了根据 PV 中的版本信息计算得到目前五个版本的大致用户比例。对于 1.0 版本，由于用户的使用比例已经较小，可以考虑让这部分用户拉取离线包信息时直接获取最新的离线包资源，而且后面的新版本发布将不再对 1.0 版本生成新的增量包。需要注意的是，新版

本覆盖率可能出现不同的分布，例如今天统计的新版本覆盖率比昨天还要低，这是很正常的，因为很多新的用户在今天的访问量减少，而老用户访问较多，但是整体上新版本覆盖率是会呈现不断上升然后开始下降的趋势，我们通常将某版本以上的覆盖率总和超过 95%（当然这个值根据不同业务的场景可以自己定义，而且尽量不要去统计单个版本的覆盖率情况）时认为是该版本覆盖完成，而将该版本发布时起到该版本以上版本覆盖率总和超过 95%的这段时间称为离线包的更新周期。

通过离线包覆盖率的统计，我们可以很清晰地了解之前版本的用户覆盖情况，也有助于进行产品的完善和改进。

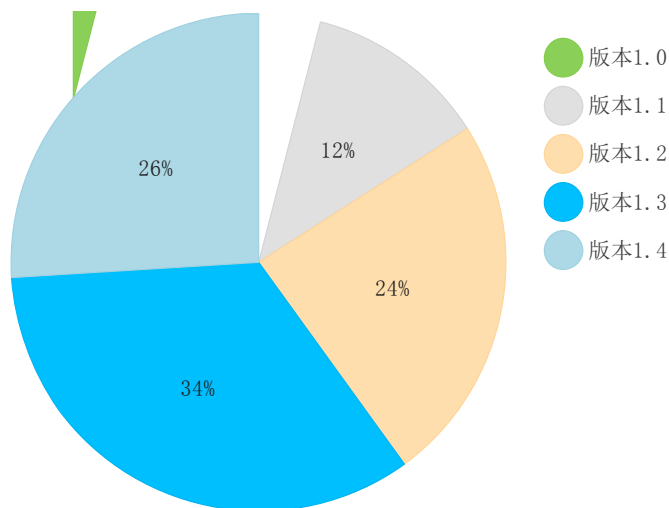


图 6-17 增量包统计示例

### 6.2.6 仍需要注意的问题

离线缓存的实现能够解决加速页面内容加载的问题，尽管如此，我们依然需要注意一些 Hybrid 应用开发时的其他问题。

#### 👉 Hybrid性能问题

了解过 Hybrid 之后，你会发现 Hybrid 应用的 WebView 存在另一个性能问题：HTML 的 DOM 渲染和操作较慢。需要注意的是，这里所说的慢是相对的，是 Native 应用 WebView 内容操作相对于移动端浏览器的内容操作来说的，即便我们把前端优化做到极致，HTML DOM 的

运行机制较慢仍是不可改变的。值得庆幸的是，目前 MNV\* 的开发模式正在改变这一局面，前面也讲到，它允许我们通过 JavaScript 来调用 Native 的原生控件生成界面，并能尽量接近 Native 应用的性能，这也在一定程度弥补了 Hybrid 应用内容渲染过程中的性能劣势。

#### 👉 前端技术栈的其他应用实现

前端技术栈还有一些其他的应用方式，如 Native 编译技术或者桌面应用开发。

Native 编译技术指的是使用前端技术编译生成 Native 应用开发工程，例如可以生成 Android 或 iOS 原生项目工程，再进行二次开发编译打包成最终的 Native 客户端版本。前端技术在桌面端应用的开发场景也在不断增多，有一些相对较成熟的框架和相关资料，实现也比较简单，有实际需求的读者可以去尝试一下。

## 6.3 本章小结

在这一章中，我们向大家介绍了前端技术在 Node 端和客户端的应用，主要包括后端渲染、前后端同构、Hybrid App 离线技术与统计等内容。掌握这些将有利于前端工程师应对更多的应用场景，在更复杂的业务应用中灵活选择实践方案。到此为止，本书要讲解的前端知识内容已经介绍完毕，在下一章中，我们将一起来看看未来前端技术的发展趋势以及如何成为一名优秀的前端工程师。



# 第 7 章

## 未来前端时代

到了这里，本书要向大家介绍的技术内容基本就讲解完了。这一章我们再一起来展望一下未来前端领域可能的发展情况。

就前端主流技术框架的发展而言，其过去几年发展极快，在填补了原有技术框架空白和不足的同时也渐渐趋于成熟。未来前端在已经趋向成熟的技术方向上面将会慢慢稳定下来，进入技术迭代优化阶段，例如语言标准、前端框架等。但这并不代表前端领域技术就此稳定了，因为新的技术方向已经出现，并在等待着下一个风口的到来。那么什么是下一个风口呢？虚拟现实？人工智能？还是其他的什么？不管未来如何，就前端应用开发方向来讲，MVVM、Virtual DOM 和同构的技术解决方案依然会延续发展一段时间，而且这段时间内前端框架技术的变化将不会像原来一样具有颠覆性。

当 MVVM、Virtual DOM 或同构等技术实践都有很成熟高效的框架和方案可以实现时，对于移动端应用，前端要重点发展的下一步可能就是 MNV\* 的原生 NativeView 开发，例如使用通用的 MNV\* 前端技术实现方案来降低移动端 Native 和前端 Web 交互的开发成本，让前端既可以通过 Native 编译开发出稳定的原生应用外壳，也可用来开发快速迭代、高性能的移动端 MNV\* 应用，最终形成一套移动端高效率的前端开发生态体系。

另一方面，新领域的 Web 化思路也会给前端带来技术革新和发展机遇，例如 Web 虚拟现实（Virtual Reality，VR）、物联网（Physical Web，将物体连入网络的一种理念）Web 化、网站人工智能等，这些方向的开发者早已跃跃欲试，目前国外也能找到少数这样的应用站点。

## 7.1 未来前端趋势

经过近几年的发展，现代前端已经革新到跨端、跨界面的阶段，主流以基于 MVVM、Virtual DOM、移动端 MNV\* 思路和前后端同构技术进行开发的项目居多，实现的方向也多种多样，这些在前面对应的章节中均有讲到。除了这些，关于未来还有一些是我们前端工程师需要了解的，下面一起来看看未来前端技术具体可能会发展成什么样。

### 7.1.1 新标准的进化与稳定

前端新标准和草案在不断更新，HTML、CSS、JavaScript 标准也在渐渐完善，尽管这些新的规范最终会淘汰旧标准，新的项目也会以最新的标准作为开发依据，但要完全停止旧标准的使用并完成企业级旧项目的升级，依然需要一段时间。例如原有 CoffeeScript 的项目不可能一次性做出迁移重构，我们的项目仍需要维护，不能脱离实际项目去谈技术，这就需要一段时间来慢慢修改；再如 Web Component 也不会马上作为唯一标准被大力推广。但可以肯定的是，新的语言或技术标准一定会被推广使用，只是还需要时间。

同时基于标准也会出现一些衍生的脚本语法和规范来适应特定的应用场景，这些非标准的规范除了解决具体业务技术问题之外，极有可能进化成下个标准的一部分或被新的标准借鉴。例如 CoffeeScript 虽然最终没有形成 JavaScript 开发标准，但 ECMAScript 6 却借鉴了其中很多优秀的特性。目前生成 Virtual DOM 的衍生脚本语法，未来也是有可能被列入到 JavaScript 标准当中的。

经过大版本的更新稳定，目前前端三层结构实现已经处于 HTML5、CSS3、ECMAScript 6+ 标准规范结合的阶段，后面标准的新变化也会越来越小。至少迄今为止，我们无法预见 HTML6 的到来，CSS4 的特性也令人担忧，ECMAScript 7 的特性更新也并不明显，这都显示出目前前端项目实践规范将会相对稳定一段较长的时间，后面的修改不会像之前一样具有颠覆性，这也是技术标准发展到一定成熟阶段必然发生的事情。

### 7.1.2 应用开发技术趋于稳定并将等待下一次革新

前端应用开发框架先后经历了 DOM API、MVC、MVP、MVVM、Virtual DOM、MNV\* 阶段，逐步解决了前端开发效率、设计模式、DOM 交互性能中存在的问题。这些问题处理完成后，相关的框架也会进入稳定发展、版本有序迭代的时期，也就是说前端的交互框架不会像以前那样变化频繁。但目前前端可能还有一件需要去做的事情，就是使用前端技术栈独立开发 Native

应用的能力,如果做到这点,前端开发者就可以结合 MNV\*开发模式独立进行 Native 应用开发并快速实现高性能的移动端应用了。因为目前的 MNV\*框架的设计实现依然依赖少数几个已有的成熟 Native 应用的运行环境,还做不到在通用的 APP 上用前端技术栈直接调用移动设备原生 API。但如果前端技术栈具备了通用的 Native 开发能力,技术上也就意味着 JavaScript 脚本(或是衍生的其他脚本)可以将任何一个普通的移动端应用编译打包成为 Native 包,并能使用 MNV\*模式直接与移动设备原生 API 进行交互。目前也有框架在做这方面的尝试,但还不理想,仍需要更多的改进完善。但无论如何,前端技术栈的 Native 开发实现技术必将成为前端技术的下一个实践核心。

### 7.1.3 持续不断的技术工具探索

前端技术效率和性能的提升当然不是仅靠前端框架就能解决的,还需要其他各方面辅助工具的支持,例如高效的调试工具、构建自动化工具、自动发布部署工具等。所以未来前端发展过程中各种高效工具仍会不断出现,解决特定场景下的问题,最后完成一个优胜劣汰的过程。

### 7.1.4 浏览器平台新特性的应用

就浏览器端应用而言,以 Chrome 为代表的浏览器版本和特性发展迭代极其迅速,经过多版本的迭代,浏览器上已经可以实现较多的增强和实用特性,例如 Web Component、Service Worker、IndexedDB、WebAssembly、WebRTC、ECMAScript 6+的支持等,但由于浏览器的种类和版本多样,我们还不能在业务中直接推广使用这些新的特性,但这些特性仍然给了我们很多实现未来技术的可能,并且未来较多技术会在这些新特性的基础上进行优化或改进产生。

### 7.1.5 更优化的前端技术开发生态

贯穿浏览器、服务端和移动端,前端正朝着多端、多技术实现的方向发展。这意味着前端这套技术栈能做的事情可能更多,涉及的平台更广。但作为整套技术开发生态的一部分,每一项技术的出现都必须要考虑开发效率、维护成本、性能、扩展性这几个方面的问题,所以寻找并发展更优的开发生态体系仍是未来前端技术的大方向,对于新技术的出现,我们也会从以下几个方面去评价它的意义。

1. 开发效率。通常提高开发效率的方式就是使用开发框架。例如 DOM 编程框架简化了脚本 API 的使用、提高了代码复用性,选择好的框架常常能让我们事半功倍。
2. 维护成本。使用框架提高了项目的开发效率,但却并不能解决代码维护的问题。这就需

要借助合适的模式来管理项目开发的代码，降低项目的维护成本，例如提取公共业务基础库、模块化、组件化等。目前最佳的实践可能就是组件化了，让业务模块的实现和管理有章可循，同时这也是 Web 标准未来发展的需要。

3. 性能。从前端开发框架的演进来说，可以总结为先专注于解决前端的开发效率问题，然后解决前端的交互性能问题，再去尝试打通 Native 开发的能力。所以性能将作为未来评价任何一个框架或技术优劣性的重要标准，同时也将是一个无法避开的永久性话题。

4. 扩展性。其实扩展性并不只是框架的方便定制和扩展特性，还要考虑是否能与原来的技术框架相兼容并解耦合。例如要使用某个新技术对原有的业务做改造，我们不可能马上就替换掉所有的业务模块，不能因为新增加的技术框架实现而导致旧的模块运行出现问题。所以在新技术的应用中，除了保证原有业务层的扩展兼容，实现功能的平滑过渡也是一个必须考虑的问题。

### 7.1.6 前端新领域的出现

除了目前浏览器、服务器、移动端上的应用开发技术变革和探索外，未来前端也会出现新的应用场景，例如 VR、物联网 Web 化、Web 人工智能等。这些虽然听着比较远，但一旦到来就会很快被使用，所以前端不仅自身发展快，推广使用也极其迅速，例如移动互联网 Web 的普及也就两三年时间。

近几年，Web VR 和物联网 Web 化的概念渐渐出现，国外甚至出现了以人工智能为支撑的 Web 应用。

首先，物联网 Web 化是随着传统软件管理的 Web 化管理而出现的，目的是为了通过 Web 手段管理传统可控的智能设备，当然这里不想去吹捧物联网的终极目标到底有多美好，只是提出了物联网 Web 化的可能性。可以肯定的一点是，人类目前所有工具类物体的 Web 化控制都是可能的，只是现在去做有一定的代价和风险存在，毕竟使用传统的软件控制到目前为止还没有遇到大的瓶颈。物联网未来的发展其实就是智能设备，通过控制这些智能设备来完成人类不容易完成的事情，如果在智能设备系统中融入人工智能的控制，这样的设备也就可以理解成机器人了。而物联网 Web 化就是通过 Web 的媒介来展示和控制这些智能设备的技术，尽管目前来看这还比较遥远。

其次，在 Web VR 方面，目前 Firefox 和 Google Chrome 也正在联合推广这一特性使浏览器支持，相信在浏览器端体验 VR 的时代也离我们不远了。不过就目前而言，软件服务的虚拟现

实技术的提升空间仍然很大，而且 VR 涉及的内容很广泛，现在涉及最多的也只是 VR 视频类，还有体感类、环境类的应用场景尚待开发。不过 Web VR 的提出无疑也为前端技术发展提供了一个可能的方向，例如目前 VR 直播也成为了一个行业内的热点技术，而且极有可能成为一种新的媒体内容表现形式出现在用户浏览器上。就目前 Web 端内容展示来说，其形式主要包括页面 3D 展示和 VR 展示两方面，3D 展示是指通过 3D 的画面来展示要显示的内容，目前浏览器上主要以 three.js 的实现为代表，而 VR 展示内容则通常是需要通过 VR 头盔配合完成页面上阅读的 3D 内容。所以现有一些例如 aframe 等 Web VR 的框架主要是在 three.js 的基础上构建的。

```
<script src="js/three.min.js"></script>
<script>
let scene, camera, renderer;
let geometry, material, mesh;

init();
animate();

function init() {
  scene = new THREE.Scene();
  camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight, 1,
10000 );
  camera.position.z = 1000;

  geometry = new THREE.BoxGeometry( 200, 200, 200 );
  material = new THREE.MeshBasicMaterial( { color: 0xff0000, wireframe: true } );

  mesh = new THREE.Mesh( geometry, material );
  scene.add( mesh );

  renderer = new THREE.WebGLRenderer();
  renderer.setSize( window.innerWidth, window.innerHeight );

  document.body.appendChild( renderer.domElement );
}

function animate() {
  requestAnimationFrame( animate );
  mesh.rotation.x += 0.01;
  mesh.rotation.y += 0.02;

  renderer.render( scene, camera );
}
</script>
```

这段代码是 three.js 提供的一个官方例子，它可以创建一个场景、一个摄像机和一个立方体，

并将立方体添加到场景中，然后通过 WebGL 来完成渲染并展示立方体的动画。对这方面有兴趣的读者可以继续进行更深入的研究。

另外，你应该听说过人工智能，不过你可能不知道 Web 和人工智能是怎么结合的。早在 2011 年就有人提出了 Web 与人工智能商业化结合的可能性，结合 Web 端的人机交互与后台的机器学习，相信这个方向未来又将催生出一批新的互联网企业。尽管目前国内还缺乏较多的应用场景，但在国外已经存在基于人工智能支撑的 Web 应用来为人们提供服务了。

可以认为我们又开始进入了一个前端技术的过渡时代，现有前端开发技术趋渐成熟，新的前端技术领域跃跃欲试，可以肯定的是物联网 Web、Web VR、人工智能必定会成为前端的下一批革命性技术。我们需要做的，仍是把握技术发展趋势，紧跟领域前进的步伐，在漫漫前端道路上继续前进。

参考资料：<https://aframe.io/>；<http://threejs.org/>。

## 7.2 做一名优秀的前端工程师

本书在前面的章节中向读者讲解了近几年来前端业界内主流的开发技术，但并不等于说读完这本书就意味着你已经掌握了前端领域的所有知识。首先，本书以原理解析和体系知识分析为主，向读者们讲解了前端的工程和技术设计思维思路，起到宏观的知识体系指导作用，而并不是深入某一方面进行剖析，主要目的是让读者了解前端技术体系的来龙去脉，知道前端每一次技术发展的原因，避免自己只在业务代码和框架中折腾。其次，前端技术的革新不会就此停止，以后依然会有更多的应用技术出现，我们仍然需要在掌握现有知识的基础上继续学习探索。无论如何，要成为一名优秀的前端工程师，我们仍需要做更多的事情。

### 7.2.1 学会高效沟通

学会高效沟通是前提，在前面的章节中也讲到过，学会使用高效的沟通方式很重要。简单来说，沟通就是通过有效的方法手段正确地表达自己或理解别人观点的一个过程。作为工程师我们不仅需要具备全面严谨的思维逻辑，良好的沟通能力也是帮助我们高效完成工作的一项必不可少的技能。

### 7.2.2 使用高效的开发工具

工欲善其事必先利其器，使用高效的工具能节省大量的开发时间。比如高效地使用 SVN/Git、编辑器辅助插件、调试工具、构建工具、测试部署工具等。这些都是实际工作中一定会接触到的，我们必须找到一个清晰的思路和便捷的途径去运用这些工具，例如快捷键的使用、构建的自动化程度等都与我们的工作效率相关。所以，笔者建议尽可能选择自动化程度更高的工具来提高效率，减少重复性工作。

### 7.2.3 处理问题方法论

作为开发者，很多时候我们除了开发新需求，设计新的项目结构，还要处理很多临时的问题。总结下来，这些问题可以归为几类：业务代码类问题、需求变更或需求风险类问题、开会等其他类问题。如果你在一个项目组里进行团队协作，那么这些事情一定会发生而且很难权衡，很占用时间，一旦处理不好，就很可能陷入困境。这里，笔者也分享一下自己处理这些问题的心得。

#### 1. 代码类问题

我想大家一定也都遇到过因为某次业务代码问题或者业务基础公共模块问题，最终导致业务数据或流程不正确，需要紧急处理。这里一般可能是通过产品经理或测试人员反馈给你的，遇到这种问题时既不能急躁，也不能盲目地去修改。第一步要先确认问题，就是弄清楚是不是真正的问题。绝大部分情况下是有问题的，但有时是因为产品经理需求的策略、设置了网络代理或测试方法环境等导致的，所以这里要稍微注意一点。要确定问题也很简单，看能否复现就知道了。如果确实有问题，那么第二步要确定是什么问题，可能是代码类问题，也可能是产品或设计考虑不全面的问题。作为开发者，我们通常只能处理可控的代码类问题，如果问题不在自己的可控范围内就要尽快沟通反馈，让相关人员做出修改，如果确认是因为自己开发引起的，第三步就要想想解决的方法了。如果问题修改很快就能解决，建议马上进行处理；如果需要较大的修改工作量，就要考虑下解决方案性价比的问题了；如果处理比较麻烦，建议通过新版本或将问题独立出来处理，当然这类情况通常会比较少。

#### 2. 需求类问题

(1) 作为开发者，我们遇到最常见的需求类问题可能就是变更了，原因是大多产品同学一般不了解技术，对需求的设计实现理解有偏差、需要调整修改。遇到这类问题也不能急躁，依然是分步来应对，首先不要直接爽快地接受需求变更，而是评估需求的等级：如果是一些小的

问题不会花太多时间，那么建议先接受确认修改；当然也不排除会带来较多额外工作量的情况，这种情况就最好重新进行需求排期，毕竟是产品同学设计时欠考虑所导致的问题，这是开发者不可控的；还有一种情况你可能也会遇到，变更的需求涉及比较细节的非核心内容，但要实现的代价较大，这种情况更多的处理方式是不接受，或降低优先级在核心功能完成后再进行处理。

(2) 除了需求变更，可能遇到的第二类问题就是应对多个需求并行的情况，甚至可能同时应对多个不同的产品经理的需求，而且每个需求都是不能马上解决的。那么这时就需要开发者做决策了，需求之间是有优先级关系的：如果一个需求不马上完成会明显导致线上大的业务流程缺陷，那就建议先从这个下手；如果几个需求都显得很紧迫，这就只能和产品经理一起讨论开发计划。一心不能两用，工作时间也不能 **double**，在给不出好建议的前提下，那就让产品经理自己决策好优先级。

(3) 无法避免需求风险管理问题。简单理解就是，需求不能按期交付，你虽然已经做好了排期管理和需求评估，但是由于各种原因（需求变更或技术方案变更）已经确定不能按期完成。那么这种情况下，你需要尽早让其他人知道实际进度。需求不能按时交付，要将风险尽早地暴露出来，千万不要等到最后告诉大家。不只是前端，各端的开发、设计人员甚至产品经理如果在团队协作时不能按时完成协作任务都应该尽早通知其他人做好风险管理工作。

### 3. 其他类问题

很多时候，原先定好的开发周期会因为各种开会让开发时间显得不足，评审会、总结会、分享会都会占用时间。要相信，这是工作常态。这种情况需要灵活处理，需求开发排期评估时预留些缓冲时间就可以解决了。

## 7.2.4 学会前端项目开发流程设计

除了完成业务开发需求，对于前端开发人员来说，另一个很重要的方面应该就是学会前端项目开发流程设计的能力。这里说的前端项目开发流程设计指的是能否快速地进行前端项目基础工程、常用业务模块以及开发流程的搭建能力。具体来说就是能否快速地设计建立一个项目开发的 **Codebase**，这里面包括前端框架选型、模块化方案、代码规范化、构建自动化、组件化目录设计、代码优化处理、数据统计、同构项目结构设计等，基于这个组建好的 **Codebase**，个人或团队其他人就能在这个基础上直接便捷、高效地开发业务模块了。

其实我们也多多少少用过一些项目开发流程工具，例如基础的构建项目就是一个简单的开发流程，但光靠它是远远不够的，需要更完善的补充。学会使用开发流程工具很简单，但要根



据不同的使用场景设计组建符合预期的开发流程工具，就要清楚开发流程中每个细节的实现。就组件化设计来说，如何选择适合自己项目的组件化管理方案就比较重要了。是按照文件管理还是按照目录管理呢？如果按目录，那么目录怎么设计？公用组件目录和非公用组件目录又该怎么区分？要考虑到的问题可能会比较多，但无论如何需要注意的一点是，在进行开发项目流程设计时，也不一定需要使用那些最新的技术和设计思路，更建议根据自己或团队的具体情况来决定。一方面，如果不这样做有可能会增加其他人对项目的上手难度，或者后期维护的成本；另一方面，可以避免过度设计，如果你的网站访问量本来就比较小，就没必要用很复杂的前、后台架构了。

### 7.2.5 持续的知识 and 经验积累管理

前端技术发展很快，在你还没有深入理解本书涉及的所有内容之前，还是需要学习的，否则你的前端知识体系应该是不完整的。即使你已经完全理解了这些内容，你依然有许多事情要做，例如你可能已经成为了项目负责人或前端管理者的角色，需要学习更多其他方面的知识，如何快速培养员工、促进团队成员高效协作等。总之，持续性的学习很重要，这不仅仅是针对前端开发人员。

作为前端开发者，学习的方式也有很多，例如看别人的技术博客、研究最新的技术方向、阅读开源代码、听技术分享会、看些书等。当然自己分享知识的方式也类似，写技术博客、上传自己的研究成果、提交自己的开源代码、去分享交流会演讲、自己写一本书，如果还不满足，那就写两本。针对技术博客的学习方式来说，推荐读者们关注一些更新较多的技术论坛或一些优秀前端团队的技术博客，常去看看总会有些收获。前端技术相关博客论坛也比较多，例如 qcon、infoq、极限前端、imweb.io、前端早读课、fex 技术周刊、w3ctech 等，除此之外你也可以加入他们的技术交流社群分享沟通。

关于前端学习，说起来简单，但实际上内容还是很多的。很多时候，我们都不知道要学什么，前端的知识很零散，涉及的方面很广，看了别人的很多博客，但都感觉只是了解点点面面，没能完全体系化地学会。本书就是要帮大家解决这个问题，书中的前几章涉及的内容通过几条主线涵盖了目前前端方面几乎所有的技术知识和设计原理，按照这些方向去研究挖掘就可以基本了解前端需要掌握的所有知识了。

### 7.2.6 切忌过分追求技术

持续性的学习积累和分享有助于我们快速地提升自己的知识面和技術能力，但是也要注意

一点，一切技术的最终目的都是为产品实现服务的。技术研究应该是在完成并希望将产品打造更好的目的上进行的，切忌过分追求技术，让自己沉迷在探索技术的道路上。我们学习技术的根本目的还是要为产品输出服务，虽然需要一段较长的时间来积累技术和提升能力，但是对技术过分地苛求反而会在产品业务的实现支持上一团糟。如果你已经进入了管理者的角色，就该快速转型为管理角色，更多从管理者的角度上为整个团队服务，不该再按照工程师的思维延续下去，对技术细节过分追求。

### 7.2.7 必要的产品设计思维

作为前端工程师，我们常常在做的事情是将产品需求和设计的功能实现，然后添加上必要的辅助数据上报，那么页面就基本可以上线了。但是，有经验的工程师开发需求时除了实现需求，做的另一件事情是思考这个需求为什么是这样的呢？是不是合理呢？如果不合理该怎么修改呢？很多时候产品人员不懂开发知识，设计细节时都是站在用户的角度上思考，功能设计都比较主观，从而可能会出现很多问题。

举个例子，某个产品交互稿希望在移动端页面最底部放置一个固定提交按钮，这样用户在手机屏幕最下面点击提交就可以了，方便用户单手操作。这个设计本身是很好的，但表单项个数是不确定的，设计最后就照做了一个宽度 80px×高度 30px 的提交按钮，设计稿很完美，表单和按钮设计都很漂亮。但如果按照设计稿完成，就会发现问题：从用户的角度上来看，按钮太小，点击不方便；从开发角度上来看，表单项较多时表单后面的填写项会被底部固定的提交按钮挡住，这时需要将提交按钮区域和表单区域用颜色区分开来才合理。所以，如果前端工程师经常做前端页面的交互实现的话，是会有一定产品设计思路的，知道一些常见需求实现的漏洞和问题，再结合自己的专业开发知识，这两个问题便可以在需求阶段就过滤掉了。

什么是产品思维？通俗地讲就是用户体验思维，或者说是将自己当成普通用户来对产品进行思考。因为一切互联网产品的设计最终都是给用户使用的，所以产品经理设计功能时也应该尽可能站在用户的角度上去思考设计，开发者也可以从一个用户的角度上去思考产品。

除此之外，作为前端工程师，强烈建议大家去看一两本关于产品经理方面的书籍。一是作为整个产品项目流程的下游实现者，有必要去了解一个互联网产品的生命周期是怎样的；二是学习一些常用的产品设计常识，在一定程度上学会分析需求的漏洞和完整性。当然这是比较理想的情况，有兴趣的读者可以去看看。

## 7.3 本章小结

这一章主要分析了未来前端技术发展的趋势，告诉大家如何成为一名优秀的前端工程师。关于如何成为一名优秀的前端工程师，个人觉得以上这些方面都是应该考虑去做的。但这仅代表个人观点，如果你有更好的建议或想法，也欢迎向笔者推荐。

未来前端技术仍会不断变化，新的领域也会出现，而我们要做的就是不忘初心，坚持前端学习的方法论，不断扩充和升级自己的知识储备，做一名优秀的前端工程师。如果你对本书的内容感兴趣或觉得还有哪些知识内容没有了解，也可以向笔者提出。希望你喜欢本书，并且期待第二版。在第二版中，笔者将会邀请更多行内有经验的开发者一起完善改进，让内容更加丰富。最后，非常感谢读完这本书，衷心希望读者都能从中有所收益。